Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería en Electrónica y Automática Industrial

Trabajo Fin de Grado

Evaluación e implementación sobre el framework Pytorch Points 3D para la extracción de información de nubes de puntos 3D usando CNNs

Autor: Adrián García Sánchez

Tutora: Noelia Hernández Parra
Cotutor: Augusto Luis Ballardini

2021

Universidad de Alcalá

Escuela Politécnica Superior

GRADO EN INGENIERÍA EN ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

Trabajo Fin de Grado

Evaluación e implementación sobre el framework Pytorch Points 3D para la extracción de información de nubes de puntos 3D usando CNNs

Autor: Adrián García Sánchez

Tutora: Noelia Hernández Parra
Cotutor: Augusto Luis Ballardini

TRIBUNAL:

Presidente: D. Ignacio Parra Alonso

Vocal 1º: D. Antonio Guerrero Baquero

Vocal 2º: Dña. Noelia Hernández Parra

FECHA: 14/09/2021

"La verdadera felicidad radica en la finalización del trabajo utilizando tu propio cerebro y habilidades."

Sōichirō Honda.

Agradecimientos

Con la culminación de este trabajo finaliza una etapa maravillosa de mi vida, llena de obstáculos y dificultades, pero también de buenos momentos. No hay espacio suficiente en el papel para agradecéroslo tanto como debería, pero permitidme dedicaros unas palabras a aquellos que habéis hecho posible todo esto.

En primer lugar se lo agradezco a todos los profesores que he tenido a lo largo de mi vida. De cierta forma todos me habéis influenciado creando en mi un afán de curiosidad que me ha llevado a seguir buscando nuevos retos.

A mi familia por haberme apoyado en todo momento. Por los ánimos en los tiempos difíciles y por ayudarme en todo lo que pudieran. Por haber creído siempre en mi y por tantas cosas más.

A mis amigos, tanto los que he conocido dentro de la universidad como los de fuera. Por todas las veces que me han ayudado y por tantos buenos momentos que hemos vivido.

Quisiera hacer una mención especial a mi tutora, Noelia Hernández, tanto por toda la ayuda que me ha proporcionado para realizar el TFG como por la paciencia y comprensión que ha tenido conmigo. Sin ella nada de esto habría sido posible. También quisiera agradecer a Ignacio Parra por su ayuda a la hora de configurar el ordenador remoto.

Y finalmente al lector, por haber dedicado parte de su tiempo a la lectura de este trabajo.

Índice general

Índice de figuras

Índice de tablas

Resumen

A lo largo de este trabajo se expondrá el estudio y trabajo realizado en torno al framework *Pytorch Points 3D*. Para ello se analizará la información publicada por el autor a través de sus medios oficiales con el fin de entender los objetivos de este entorno de trabajo. Una vez visto el alcance se presentará el estudio en profundidad realizado del framework, acompañándolo con varios ejemplos propuestos por el autor. Estos ejemplos han sido modificados y extendidos para poder extraer de ellos la máxima información posible, y a su vez habilitándolos para ser usados en distintos entornos de ejecución. Posteriormente se mostraran diversos experimentos realizados con el fin de comprender mejor las posibilidades que nos ofrece dicho framework. Primero se observarán varias pruebas realizadas sobre los hiperparámetros para hallar las configuraciones óptimas, y posteriormente se expondrá un ejercicio basado en la técnica de *transfer learning*.

Palabras clave: Redes neuronales convolucionales, Pytorch Points 3D, Nubes de puntos, Python.

Abstract

Throughout this work, the study and work carried out around the framework *Pytorch Points 3D* will be exposed. For this, the information published by the author will be analyzed in order to understand the objectives of this work environment. Once the scope is seen, the in-depth study carried out of the framework will be presented, accompanied by several examples proposed by the author. These examples have been modified and extended to be able to extract as much information as possible from them, and thus enabling them to be used in different execution environments. Subsequently, several experiments will be shown in order to better understand the possibilities offered by this framework. We will start by experimenting with the hyperparameters to find the optimal settings, and then an exercise based on the *transfer learning* technique will be presented.

Keywords: Convolutional neural network, Pytorch Points 3D, Point clouds, Python.

Resumen extendido

Con elaboración de este *trabajo de fin de grado* se pretende comprender el funcionamiento del framework *Pytorch Points 3D*, para poder usarlo en tareas de extracción de información procedente de nubes de puntos tridimensionales al trabajar con redes neuronales convolucionales.

Para ello se analizará la información expuesta por su autor a través de su Github oficial, en el cual define las aspiraciones de dicho framework. Desde el Github es por donde el autor muestra todo su contenido, pudiendo encontrar tanto las distintas partes por las que está compuesto, como las redes integradas o las bases de datos con las que trabaja. El autor menciona que este framework pretende convertirse en el núcleo común para trabajar con redes neuronales convolucionales en tareas relacionadas con nubes de puntos.

Seguidamente se mostrará al estudio en profundidad realizado sobre el framework. En esta etapa se acompañará a las explicaciones con una serie de ejemplos subidos por el autor, gracias a los cuales se logra entender el funcionamiento del framework. Se explicarán también las modificaciones que han de llevarse a cabo en los mismos para poder trabajar con ellos desde Google Colab, así como diversas mejoras que se han implementado con el fin de analizar mejor todo lo que nos ofrecen.

Posteriormente se expondrá una etapa de experimentación para la cual se cuenta con un ordenador remoto con capacidad de procesamiento de datos por GPU. A lo largo de esta etapa se llevarán a cabo diversos experimentos basados en las conclusiones obtenidas en el capítulo anterior, con el fin de trabajar de primera mano con el material proporcionado. Primero se mostrará un experimento realizado sobre los *hiperparámetros* para observar el efecto que estos tienen en el entrenamiento y encontrar los valores óptimos de los mismos. Después se expondrá un ejercicio basado en la técnica de *transfer learning*, divido en varias secciones a través de las cuales se observarán los pasos necesarios para lograr entrenar una misma red en diferentes bases de datos. Estas secciones sirven a su vez para reforzar varios aspectos en los que se entrará más en profundidad, como puede ser el análisis de las entradas y salidas de la red y las fases de cargado de datos.

Finalmente se presentarán las conclusiones obtenidas sobre el framework, en donde se mostrarán las posibilidades reales que se han encontrado, así cómo sus limitaciones. Además se realizará una reflexión sobre los resultados obtenidos en los diferentes experimentos, y se mostrarán las conclusiones finales a las que se han llegado.

Capítulo 1

Introducción

El pesimista se queja del viento; el optimista espera
que cambie; el realista ajusta las velas.

William George Ward

¿Alguna vez se han parado a pensar en lo mucho que ha avanzado la humanidad en los últimos años? Los primeros homo sapiens surgieron hace aproximadamente unos 200.000 años, pero no fue hasta el 4500 a.C. que apareció la rueda y hasta que en 1886 no se acabó de desarrollar el motor de combustión interna, los "coches" eran tirados por caballos. Desde entonces se han desarrollado innumerables avances en este campo como el conocido motor eléctrico, pero es que a día de hoy ya existen vehículos capaces de conducir sin necesidad de la intervención humana, e incluso se están desarrollando proyectos para conseguir que estos vehículos se puedan desplazar sin tocar el suelo. De igual manera podrían pensar en el campo de las telecomunicaciones, donde a finales del siglo XIX solo algunos podían comunicarse por medio de puntos y rayas, y desde finales del siglo pasado ya todos éramos capaces de hablar entre nosotros desde cualquier parte del mundo. A día de hoy el compartir palabras se nos ha quedado pequeño, y no contentos con poder mandar imágenes y vídeos se está empezando a trabajar en enviar información tridimensional. Podríamos estudiar otros campos tecnológicos y su desarrollo a través de los años, pero si se paran a pensar un poco más a fondo se pueden percatar de que muchos de ellos convergen en algo común. Una tecnología revolucionaria capaz de llevar a la humanidad al siguiente nivel conocida como *Inteligencia Artificial*.

El origen de esta tecnología se remonta a 1943, con la aparición de la lógica umbral de manos de McCulloch y Pitts [1]. Este modelo matemático intentaba replicar el comportamiento de una neurona biológica haciendo uso de algoritmos, pero entonces se planteó de manera teórica, pues no se encontró ninguna aplicación práctica. Varios años más tarde, en 1958 Frank Rosenblatt presentó el perceptrón [2], un modelo de neurona basada en la lógica umbral que era capaz de realizar operaciones simples como el reconocimiento de algunos patrones. Desde entonces se empezaron a desarrollar técnicas para explotar las posibilidades que nos ofrecía esta nueva tecnología cómo la introducción de neuronas multicapa en 1969 por Minsky y Papert [3] y el desarrollo del algoritmo de "backpropagation" [4].

Entre finales del siglo XX e inicios del siglo XXI el sector se vio sumergido en lo que a día de hoy se conoce como "el invierno de la inteligencia artificial". Durante este periodo de tiempo la capacidad

computacional no era suficiente para avanzar al ritmo que se tenía pensado, lo cual sumado a otros factores como el furor por otras tecnologías emergentes como la "burbuja punto-com" y los malos resultados recogidos en el informe Lighthill [5] provocó un severo recorte en su financiación, y por tanto un estancamiento general en los estudios.

A pesar de estos contratiempos que llevaron a una caída en la reputación de la inteligencia artificial, se continuaron desarrollando nuevas propuestas, y para inicios del siglo XXI, cuando la tecnología era más avanzada, se empezaron a ver resultados que devolvieron algo de esperanza al sector. En el año 2010 la inteligencia artificial volvió a ganar interés por parte de la comunidad de investigación, lo que llevó a un auge en la financiación e inversión en el sector. Desde entonces el avance en este campo de estudio ha sido exponencial, consiguiendo a día de hoy redes neuronales capaces de llevar a cabo desde tareas de procesado de imágenes como AlexNet (2012) [6] o GoogleNet (2014) [7] a incluso poder ejecutar la toma de decisiones en una partida de ajedrez como AlphaZero (2017) [8] o escribir artículos y mantener conversaciones como GPT-3 (2020) [9]. La inteligencia artificial está cada vez más presente en nuestro día a día, es un campo que está en continuo desarrollo con un sinfín de aplicaciones, por eso merece la pena su estudio.

1.1 Objetivos

Este TFG se centra en el framework "Pytorch Points 3D" [10], el cual ha sido creado con el fin de proporcionar soporte para poder realizar diferentes tareas de extracción de información proveniente de nubes de puntos haciendo uso de distintas redes neuronales. Obtener información de nubes de puntos puede resultar muy útil en multitud campos, pero en especial puede ser de utilidad para ejercicios de navegación. Por ejemplo, en el caso de la conducción autónoma se puede obtener información del entorno fácilmente empleando dispositivos como el *lidar*, lo cual puede ayudar a la hora de detectar otros coches o peatones que se encuentren en nuestro camino.

Para estudiar el framework y poder trabajar con el correctamente se plantean los siguientes objetivos:

- Estudio y comprensión de la documentación proporcionada por el creador del framework.
- Estudio y desarrollo de técnicas de programación en Python.
- Adaptación a los diferentes entornos de trabajo como Google Colab y un ordenador remoto con capacidad de procesamiento en GPU.
- Comprensión y adaptación de los ejemplos proporcionados en la documentación así como el estudio de sus parámetros.
- Análisis de los datos de entrada para poder trabajar con distintas bases de datos.
- Adaptación del framework para resolver nuevos problemas mediante el uso de *transfer learning*.
- Análisis de los resultados obtenidos.

1.2 Organización de la memoria

El objetivo principal del trabajo es estudiar el funcionamiento de este framework, y realizar varias pruebas en las que podamos mostrar algunas de sus aplicaciones. Para ello se divide la memoria en los siguientes apartados:

1. *Estado del arte*: Aquí se hará tanto una introducción a las redes neuronales convolucionales como a Pytorch Points 3D, explicando las partes más teóricas de ambos para poner en contexto al lector.

2. *Estudio del framework*: En este apartado se recoge toda la información extraída del framework a partir de los ejemplos que nos proporciona. Se analizará como trabajar con él en diferentes ejercicios y las adaptaciones que hay que realizar para mejorar su funcionamiento. También se profundizará en ciertos puntos que resultarán de interés en el futuro.

3. *Experimentación con el framework*: En este apartado se realizarán varias pruebas utilizando el framework para comprender sus funcionalidades y limitaciones, así como se estudiará el comportamiento de los diferentes parámetros que encontramos a lo largo de las pruebas.

4. *Conclusiones generales y trabajos futuros*: Por últimos se incluirá una conclusión de lo estudiado y se propondrán varias aplicaciones que se pueden derivar de este trabajo.

Capítulo 2

Estado del arte

No se puede desatar un nudo sin saber cómo está hecho.

Aristóteles

En este capítulo se realizará una breve introducción a las redes neuronales, su evolución y los principios teóricos en los que se basa su funcionamiento. También presentaremos el framework *Pytorch Points 3D* y hablaremos de las posibilidades que nos ofrece.

2.1 Redes neuronales

Como se menciona en la introducción la tecnología con la que se va a trabajar es relativamente reciente, pero en pocos años su desarrollo ha sido exponencial, por lo que para entenderla correctamente debemos empezar por entender sus orígenes.

2.1.1 Unidad lógica de umbral (LTU)

La **unidad lógica de umbral** [1] fue la propuesta de McCulloch y Pitts en 1943 para replicar el comportamiento de una neurona biológica. El principio de esta se centra en la **función umbral** (threshold function), una función booleana monótona representada de la siguiente forma:

$$f(x) = \begin{cases} 1 & si \quad \sum \omega_i x_i \geq t \\ 0 & si \quad \sum \omega_i x_i \leq t \end{cases}$$

Figura 2.1: Función lógica de umbral.

Esta propuesta surge del estudio de las neuronas en base a una serie de criterios expuestos en el paper de McCulloch y Pitts [1], como que las neuronas tienen un funcionamiento del estilo "todo o nada", de ahí surgió la función umbral, la cual vale 1 en caso de que el valor de entrada supere cierto valor umbral, y 0 en caso contrario, simulando este tipo de comportamiento.

También a lo largo del escrito los autores hacen referencia a una serie de valores que acompañan a las neuronas y que determinan su respuesta, así como a la interacción que sufren las neuronas entre sí, representado en la figura 2.2.

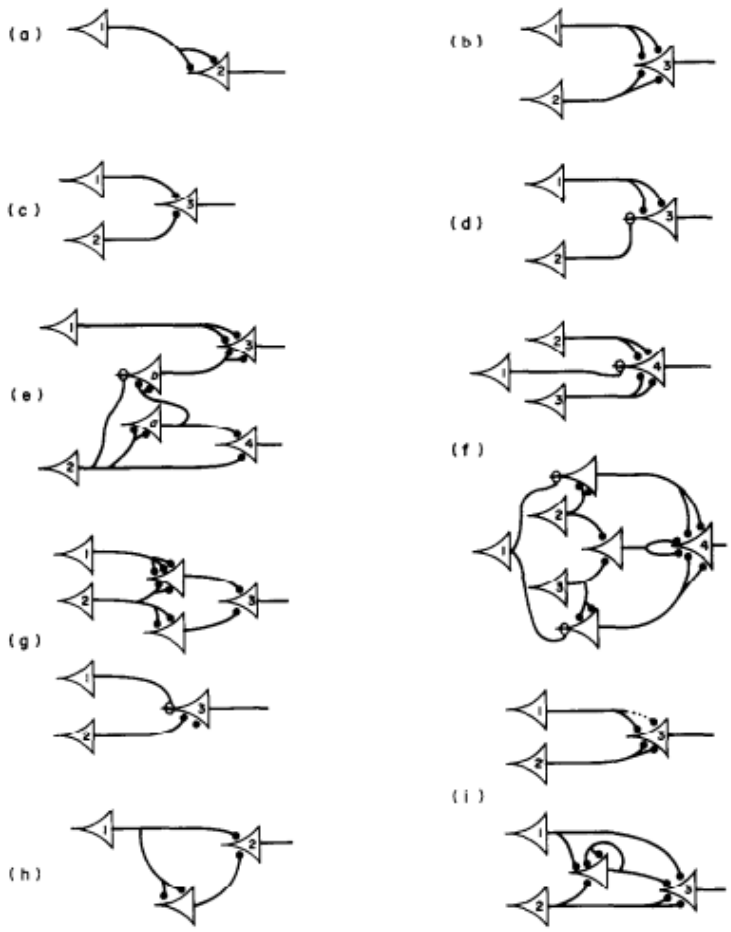
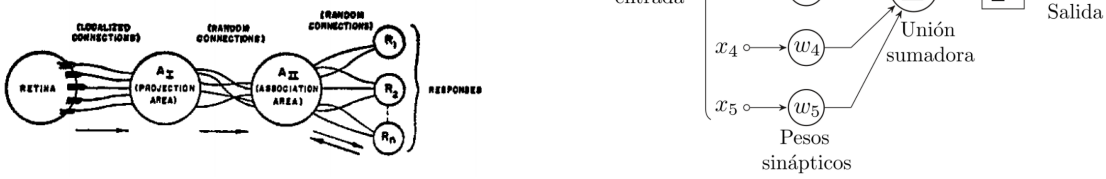


Figura 2.2: Diagrama que muestra la relación entre las neuronas [1].

Sin darse cuenta los autores introdujeron de manera teórica muchos conceptos que se usarían más adelante; como es el caso de los pesos, representados por esos números que acompañan a las neuronas o las funciones de activación, como la función umbral que sería la primera de ellas en aparecer. Debido a las limitaciones de la época todo esto se quedó de manera conceptual ya que la tecnología que había no permitía seguir desarrollando esta teoría.

2.1.2 Perceptrón

En 1958 apareció el **perceptrón** [2] de la mano de Frank Rosenblatt (figura 2.3). Este se trata de un modelo matemático inspirado en la lógica umbral [1], el cual pese a estar muy limitado, ya era capaz de realizar operaciones simples tales como las funciones lógicas AND y OR.



(a) Representación de un perceptrón [2].

(b) Diagrama de un perceptrón con cinco señales de entrada

Figura 2.3: A la izquierda primera representación de la estructura de un perceptrón expuesta en [2] y a la derecha una representación actual de un perceptrón.[1]

Esta neurona artificial toma un vector binario de entrada, representados en la figura 2.3 con la variable x_i que genera un único valor de salida, representado por la variable y. Para calcular la salida cada valor del vector de entrada es multiplicado por un número llamado peso y representado por la variable ω_i, que representa lo significativa que es esa entrada para una función específica. Seguidamente se suman todos los resultados y se llevan a una función de activación, que determina el valor final de salida. Matemáticamente se puede ver el funcionamiento del perceptrón de la siguiente forma:

$$y = \sum_{i=1}^{n}(x_i * w_i) + b_i \tag{2.1}$$

En la fórmula se añade el parámetro b_i que representa un valor de offset que se puede asociar con cada una de las neuronas.

Finalmente el valor obtenido ha de ser evaluado con una función de activación. Esta devuelve un valor binario en función del valor de entrada, generando así ese comportamiento de *todo o nada* explicado en el apartado de la lógica umbral. Existen muchos tipos de funciones de activación, pero las más reconocibles son las representadas en la figura 2.4.

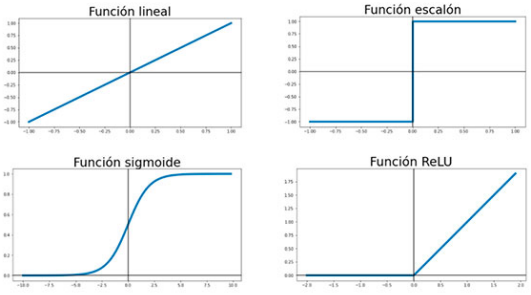


Figura 2.4: Funciones de activación. [2]

[1] Imagen actual del perceptrón extraída de https://es.wikipedia.org/wiki/Perceptrón

[2] Imagen extraída de *https://www.futurespace.es/redes-neuronales-y-deep-learning-capitulo-2-la-neurona/*

- **Función lineal**: Se trata de la función identidad. Esta función hace que la salida sea igual a la entrada. Si se usa esta función, entonces la neurona se comporta exactamente igual que una regresión lineal.
- **Función escalón**: Útil cuando la salida es categórica y se pretende clasificar, pero en la práctica no es muy utilizada por las dificultades a la hora de trabajar con su derivada.
- **Función sigmoide**: Muy útil ya que consigue que los valores muy grandes converjan a 1 y los muy pequeños a -1, por lo que sirve para representar probabilidades. Su único inconveniente es que su rango no está centrado en el origen, por ello a veces es recomendable sustituirlo por la función tangente hiperbólica.
- **Función ReLU (unidad rectificada lineal)**: Se comporta como una función constante para valores negativos y como una función lineal para valores positivos. Es una de las funciones de activación más utilizadas.

En la práctica se demostró que que este modelo estaba muy limitado. Solo era capaz de resolver aquellos problemas cuyos elementos dibujados en un plano pudiesen ser separados por un hiper-plano que dividiese las soluciones en dos zonas, una con los elementos 'deseados' y otra con los 'no deseados'. Por ejemplo este modelo era capaz de simular el comportamiento de las puertas lógicas AND y OR ya que al dibujar su funcionamiento sobre un plano se puede ver como los resultados se pueden separar usando una sola línea recta, tal y como se muestra en la figura 2.5. Sin embargo no era capaz de simular el comportamiento de la puerta XOR, ya que esta al ser más compleja requiere de más de un hiper-plano para separar sus soluciones.

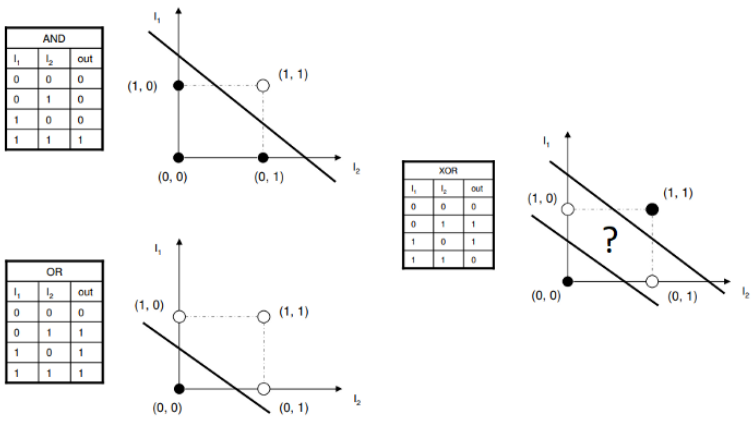


Figura 2.5: Funcionamiento de las puertas lógicas. [3]

2.1.3 Perceptrón multicapa (red neuronal)

Con el fin de mejorar la idea propuesta por Rosenblatt, Marvin Minsky y Seymour Papert publicarón en 1969 un libro donde mostraban un nuevo modelo capaz de resolver aquellos problemas a los que el perceptrón no daba solución, el **perceptrón multicapa** [3], representado en la figura 2.6.

[3]Imagen extraída de *https://medium.com/@lucaspereira0612/solving-xor-with-a-single-perceptron-34539f395182*

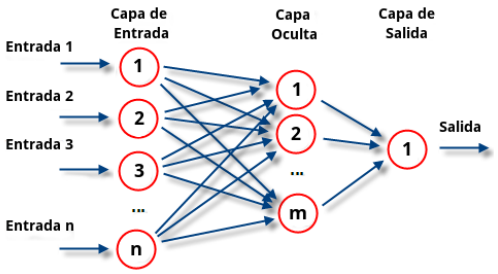


Figura 2.6: Estructura perceptrón multicapa. [4]

Al aumentar el número de capas, este modelo era capaz de resolver problemas más complejos. De este modo, por ejemplo el problema de la XOR encontró una solución ya que con dos capas obtenemos los dos hiper-planos necesarios para poder dividir sus elementos (deseados y no deseados). De esta forma se encontró una solución a ciertos problemas no lineales con los que el perceptrón simple no podía lidiar. Ahora se presentaba un nuevo problema ya que no existía un mecanismo automático para adaptar el valor de los pesos de las capas ocultas, lo que hacía que fuese imposible trabajar con ello.

No fue hasta el año 1986 que Rumelhart junto con otros autores presentaron la **Regla Delta Generalizada** [4] proporcionando así un método para adaptar los pesos propagando los errores hacia atrás, el cual apodaron como ***backpropagation***. Este método consiste en comparar la salida obtenida al meter una entrada controlada con la salida ideal deseada, calcular la diferencia (error) aplicando la siguiente fórmula:

$$E(X) = \frac{1}{2}\sum_{i=1}^{n}(y_{deseada} - y_{obtenida})^2 \tag{2.2}$$

Y finalmente actualizar el valor de los pesos de las capas *'ij'* de la siguiente forma:

$$\omega_{ij}(t+1) = \omega_{ij}(t) + \gamma\frac{\delta E(t)}{\delta\omega_{ij}(t)} \tag{2.3}$$

Donde ω_{ij} representa el peso que une la capa i con la capa j, $\frac{\delta E(t)}{\delta\omega_{ij}(t)}$ representa el gradiente del error evaluado respecto del peso *'ij'* y γ representa el *learning rate*, un hiper-parámetro con el que podemos ajustar la velocidad de modificación de los pesos.

2.1.4 Red neuronal convolucional

Más adelante se desarrollaron las **redes neuronales convolucionales (CNN)**. Este tipo de red es una variación de un perceptrón multicapa orientado a trabajar con matrices, lo cual hace que sea idóneo para trabajar con imágenes. Un ejemplo de este tipo de estructuras se puede observar en la figura 2.7.

[4]Imagen extraída de *https://es.wikipedia.org/wiki/Perceptrón_multicapa*

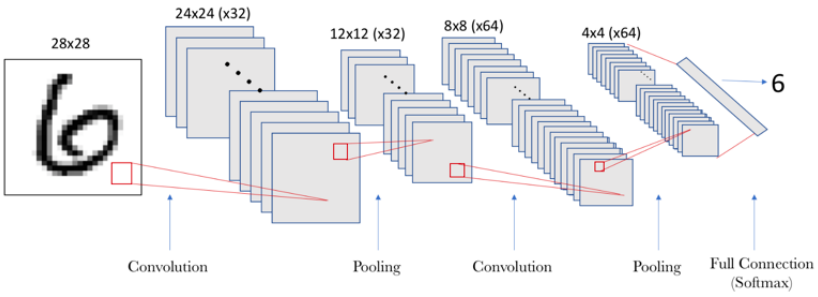


Figura 2.7: Estructura de una red neuronal convolucional. [5]

Este tipo de redes se caracterizan por estar diferenciadas en dos partes. Las primeras capas están constituidas por una serie de filtros convolucionales y neuronas de reducción de muestreo que se encargan de extraer las características de la entrada. Las últimas capas suelen estar constituidas por neuronas completamente conectadas que sirven para realizar las tareas de clasificación final.

Neuronas convolucionales

Estas neuronas se encuentran en la fase de extracción de características. Se tratan de procesadores en matriz que realizan una operación sobre los datos de la imagen que pasan por ellas. La salida se puede calcular como:

$$Y_j = g(b_j + \sum_i (K_{ij} \bigotimes Y_i)) \tag{2.4}$$

Donde la salida de una neurona j es una matriz que se calcula con una combinación lineal de las salidas de las neuronas de la capa anterior Y_i operadas por un núcleo convolucional K. Este valor es sumado a una influencia b y luego pasada por una función de activación no lineal $g()$.

Este operador realiza un efecto de filtrado sobre la imagen de entrada con un núcleo previamente entrenado, lo que transforma los datos de manera que ciertas característica se vuelven más dominantes.

Neuronas de reducción de muestreo

Estas neuronas realizan una reducción de la resolución, consiguiendo un mayor campo de actividad en la imagen generada. Normalmente se llevan a cabo con operaciones tipo max-pooling, las cuales generan una imagen reducida de la original a partir de una ventana de muestra que es pasada por la imagen de entrada píxel a píxel, generando así una imagen de salida en la que el valor de cada píxel corresponde con el máximo de la ventana para cada región. En la figura 2.8 se puede ver un ejemplo en el que una imagen 4X4 es reducida a otra 2X2 aplicando un núcleo de muestreo. Estos núcleos pueden ser de multitud de tipos, y son los que determinan que características se resaltan. Por ejemplo, en el caso de la figura la función del núcleo sería resaltar las líneas verticales.

[5] Imagen extraída de *https://torres.ai/deep-learning-inteligencia-artificial-keras/*

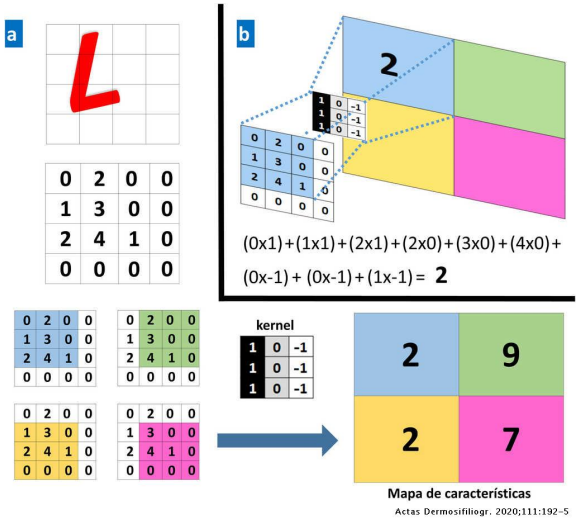


Figura 2.8: Ejemplo del funcionamiento de una neurona de reducción de muestreo. [6]

Neuronas de clasificación

Tras las fases de extracción de características, los datos son llevados a una serie de capas completamente conectadas que realizan la tarea de clasificación. Estas funcionan igual que el perceptrón multicapa, y tienen la tarea de clasificar las imágenes en función de las características generadas en las anteriores etapas.

Al juntar todas las capas se obtienen redes capaces de llevar a cabo tareas de extracción de información complejas, como es el caso del ejemplo de la figura 2.9 donde se puede observar simplificada la estructura de una red usada para la detección de problemas en la piel. En ella se puede notar como una imagen es transformada a través de capas de convolución y muestreo para la extracción de sus características, y después estas son analizadas por una serie de capas totalmente conectadas que determinan la salida. La salida es mostrada en el recuadro amarillo, donde vemos que todas las posibilidades son listadas en función de su probabilidad.

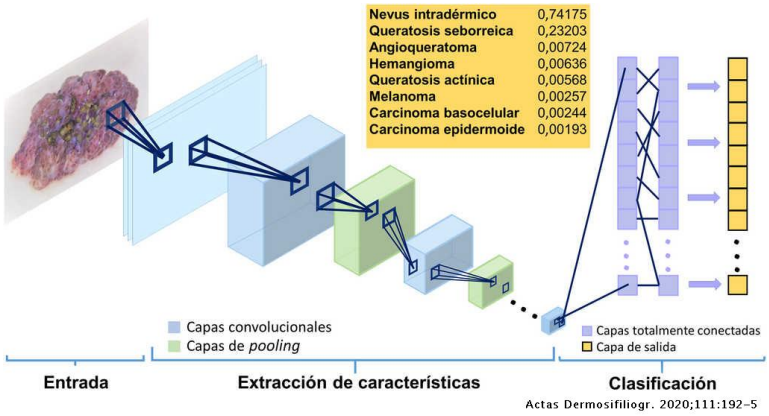


Figura 2.9: Ejemplo del funcionamiento completo de una red convolucional. [5]

[6] Imágenes extraídas de *https://www.actasdermo.org/es-deep-learning-dermatologia-articulo-S000173101930393X*

2.2 Pytorch Points 3D

Pytorch Points 3D [10] es un framework de código abierto diseñado por Thomas Chaton, Nicolas Chaulet, Sofiane Horache y Loic Landrieu para facilitar el uso de redes neuronales convolucionales que trabajan con datos 3D. Este framework da soporte para realizar múltiples tareas en entornos tridimensionales tales como la detección de objetos o la segmentación panóptica, con diferentes tipos de arquitecturas como KPConv, PointNet++ o con un diseño propio. El objetivo de los creadores era el diseñar un entorno de trabajo estándar sobre el que cualquier persona pueda realizar modificaciones fácilmente para alcanzar diferentes objetivos. De esta forma podemos configurar, por ejemplo, PointNet++ que es una arquitectura diseñada principalmente para tareas de segmentación y clasificación de forma que lleve a cabo tareas de detección.

El framework esta basado en *Pytorch Geometric* [11] y en *Facebook Hydra*, y está preparado para trabajar con el lenguaje de programación *Python*. Para su uso se requiere de una GPU con una versión de CUDA 10 o mayor, una versión de Python que como mínimo ha de ser la 3.7 y de Pytorch mayor que 1.7.

Actualmente el framework está preparado para poder trabajar en las siguientes tareas:

- **Clasificación (classification)**: Consiste en identificar y diferenciar diferentes tipos de objetos.

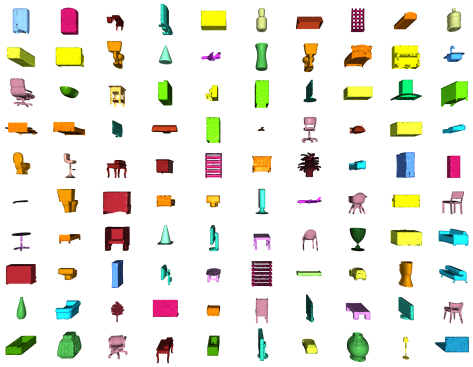


Figura 2.10: Clasificación en Pytorch Points 3D [10].

- **Segmentación semántica (segmentation)**: Consiste en identificar las distintas partes que componen un objeto.

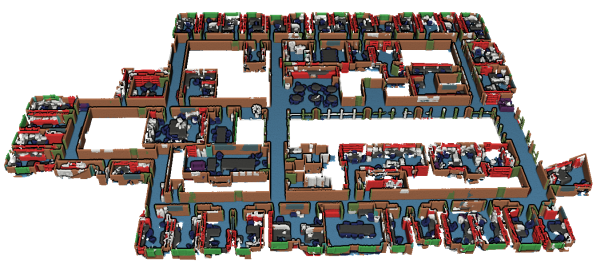


Figura 2.11: Segmentación en Pytorch Points 3D [10].

- **Detección de objetos (object detection)**: Consiste en identificar los diferentes objetos presentes en un mismo escenario.

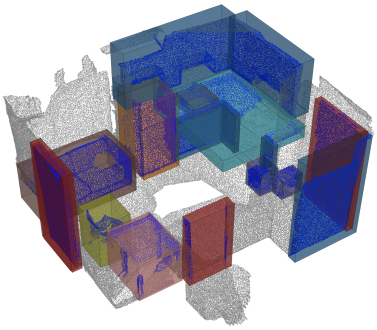


Figura 2.12: Detección de objetos en Pytorch Points 3D [10].

- **Segmentación panóptica (Panoptic segmentation)**: Se trata de una combinación entre la segmentación semántica (distinción de categorías de alto nivel significante, usualmente objetos) y la instanciada (distinción de individuos de una misma categoría) de forma que podemos identificar en una imagen la clase a la que pertenece cada píxel y su número de instancia.

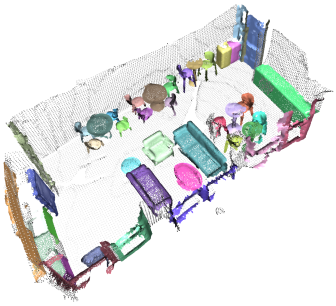


Figura 2.13: Segmentación panóptica en Pytorch Points 3D [10].

- **Registro (registration)**: Proceso mediante el cual dos o más volúmenes, con sus correspondientes características, se alinean en la misma coordenada espacial.

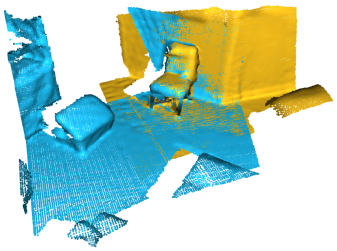


Figura 2.14: Registro en Pytorch Points 3D [10].

Para facilitar el trabajo a la hora de entrenar las distintas redes, el propio framework habilita a los usuarios de forma nativa para poder trabajar con multitud de bases de datos. Actualmente las bases de datos disponibles son las siguientes:

- **Scannet** [12]: Base datos que suministra nubes de puntos de más de 1500 entornos tridimensionales (y hasta 2,5 millones contando distintos puntos de vista) de distintas estancias del hogar. Se sugiere esta base de datos para tareas de segmentación por partes, segmentación panóptica y de detección de objetos.

Figura 2.15: Ejemplo de espacios reconstruidos en ScanNet [12].

- **S3DIS** [13]: Base de datos que suministra nubes de puntos de espacios 2D, 2.5D y 3D. Contiene más de 70,000 imágenes de diferentes salas de 3 edificios completamente diferentes y casi 1,500 imágenes equirectangulares (proyección cilíndrica). Se sugiere utilizar esta base de datos para realizar tareas de segmentación por partes, segmentación panóptica y de detección de objetos.

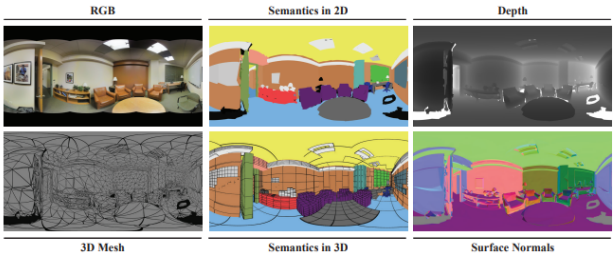


Figura 2.16: Ejemplos de datos 2D-3D-Semantic en S3DIS [13].

- **Shapenet** [14]: Base de datos de nubes de puntos 3D de modelos de diferentes categorías. Contiene más de 50,000 modelos 3D de 55 categorías diferentes como barcos, aviones o gorras. Se sugiere utilizar esta base de datos para realizar tareas de segmentación.

Figura 2.17: Ejemplo de de modelos disponibles en Shapenet [14].

- **SemanticKitti** [15]: Base de datos de nubes de puntos obtenidos desde un vehículo en movimiento, utilizando un dispositivo LiDAR. Contiene 22 secuencias con más de 40,000 escaneos. Se sugiere utilizar esta base de datos para realizar tareas de detección de objetos y segmentación panóptica.

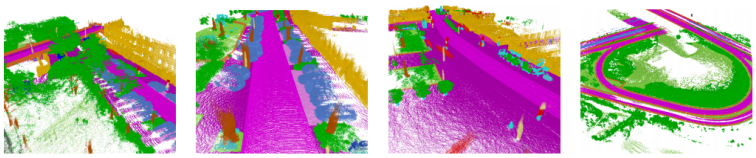


Figura 2.18: Ejemplos de nubes de puntos disponibles en SemanticKitti [15].

- **3DMatch** [16]: Base de datos de nubes de puntos de diferentes entornos domésticos. Contiene más de 20,000 estructuras 2D/3D complementadas con datos adicionales como las coordenadas de los píxeles y la cámara o la matriz intrínseca de la propia cámara que pueden ser usadas en otro tipo de tareas dentro del campo de la visión artificial. Se sugiere utilizar esta base de datos para realizar tareas de registro.

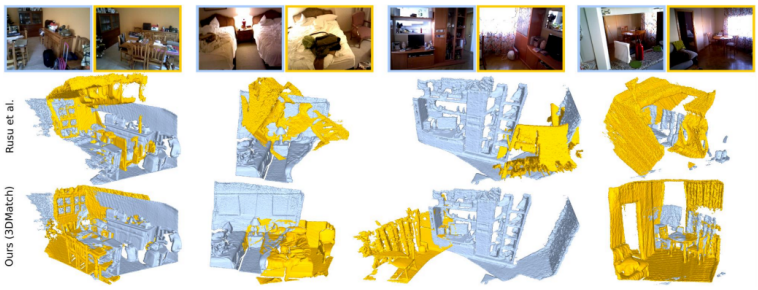


Figura 2.19: Ejemplo de estructuras disponibles en 3DMatch [16].

- **The IRALab Benchmark** [17]: Benchmark que reúne nubes de puntos extraídos de las siguientes bases de datos: ETH [18], Canadian Planetary Emulation Terrain 3D Mapping [19], TUM Vision Ground RGBD [20] y KAIST Urban [21]. Se sugiere para tareas de registro.

- **Kitti odometry** [22]: Benchmark que consiste en 22 secuencias estéreo tipo SLAM tomadas desde una cámara instalada en un vehículo. Se recomienda para tareas de registro.

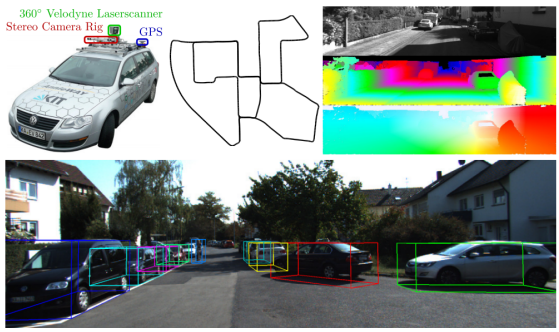


Figura 2.20: Ejemplo de datos disponibles en Kitti odometry [22].

- **ModelNet** [23]: Base de datos que reúne nubes de puntos 2.5D y 3D de diferentes objetos tridimensionales. Tiene modelos CAD de 10 categorías diferentes como sillas o mesas, divididas a su vez en otras 10 subcategorías. Se recomienda para tareas de clasificación.

Figura 2.21: Ejemplo de nubes de puntos disponibles en ModelNet [23]

Aunque puedes usar cualquier arquitectura que diseñes tu mismo, el propio framework está preparado para trabajar fácilmente con los siguientes módulos:

2.2.1 KPConv

Kernel Point Convolution (KPConv) [24] es una red neuronal convolucional diseñada por Hugues Thomas, Charles R. Qi, Jean-Emmanuel Deschaud, Beatriz Marcotegui, François Goulette y Leonidas J. Guibas en 2019 que propone una arquitectura basada *kernel points*, consiguiendo un diseño capaz de operar con nubes de puntos sin necesidad de un intermediario. Esta red consta de 5 capas de clasificación, las cuales están compuestas por dos bloques convolucionales. Estos bloques tienen un diseño parecido al *bottleneck* de los bloques ResNet, pero reemplazando la convolución de imágenes. Cuenta a su vez con *batch normalization* y con funciones de activación tipo ReLu. Tras la última capa las características son agregadas por una capa *average pooling* y finalmente procesadas por una capa totalmente conectada con función de activación tipo *softmax*.

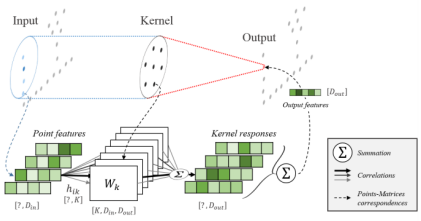


Figura 2.22: Estructura de la red KPConv [24].

La forma de trabajar de esta red es en base a las distancias entre los puntos. Para ello toma los “radios vecinos” como entrada y los procesa usando una serie de pesos separados en pequeños conjuntos de “kernel points” definidos en el espacio euclídeo.

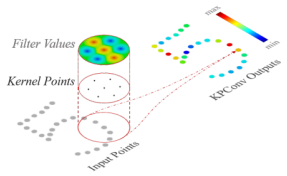


Figura 2.23: Ilustración del funcionamiento de KPConv [24].

Además nos permite trabajar con dos tipos de configuraciones distintas para los núcleos:

- **Rígida**: en este tipo de configuración se requiere de una disposición regular de los núcleos, por lo que necesita de una organización constante para ser eficiente. Se ha demostrado que esta configuración da mejores resultados para tareas simples.
- **Deformable**: en este tipo de configuración no se requiere de una disposición fija ya que los propios núcleos aprenden a adaptarse a la geometría local. Esta configuración da mejores resultados en tareas complejas.

Los autores nos presentan KPConv como una red neuronal convolucional capaz de desempeñar diferentes tareas, pero especializada en la clasificación y segmentación.

2.2.2 PointNet++

PointNet++ [25] es una red neuronal convolucional creada por Charles R. Qi, Li Yi, Hao Su y Leonidas J. Guibas en 2017 como respuesta ante las limitaciones que tenía PointNet [26]. En esta nueva versión se implementan mejoras con el fin de respetar la localización espacial de los conjuntos de puntos. Para ello se introduce una red neuronal jerárquica que aplica PointNet varias veces sobre una muestra de los puntos de entrada.

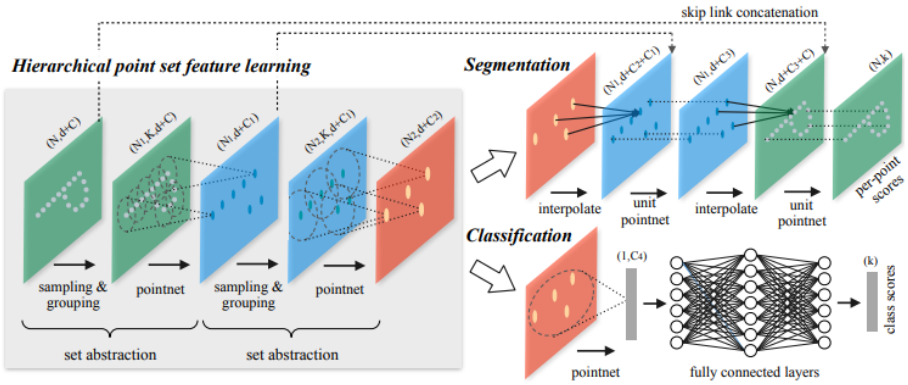


Figura 2.24: Estructura de PointNet++ [25].

PointNet++ se puede ver como una extensión de PointNet. Mientras que la primera usaba una simple capa maxpooling, la nueva arquitectura añade una estructura jerárquica compuesta por un conjunto de niveles abstractos donde cada nivel se compone de tres capas:

1. Una **capa de muestreo** que selecciona un conjunto de puntos de la entrada y selecciona los centroides.

2. Una **capa de agrupamiento** que construye una región local en la que se buscan los puntos vecinos alrededor de los centroides.

3. Una **capa PointNet** que codifica los patrones de la región local en vectores de características.

Por último, en función de la tarea a desempeñar se acaban las últimas capas de la manera más óptima, ya sea con un conjunto de capas totalmente conectadas en caso de querer realizar una clasificación o con capas de interpolación si queremos realizar una segmentación. De esta forma los autores nos proponen esta arquitectura capaz de manejar problemas con entradas de puntos no uniformes, lo que habilita a la red a conseguir resultados frente a entradas de tipo nubes de puntos tridimensionales. Esta red está pensada principalmente para llevar a cabo tareas de clasificación y segmentación.

2.2.3 Relation-Shape CNN

Relation-Shape Convolutional Neural Network (RS-CNN) [27] es una red creada por Yongcheng Liu, Bin Fan, Shiming Xiang y Chunhong Pan en 2019 con el objetivo de analizar nubes de puntos de configuraciones irregulares. Para conseguirlo proponen un conocimiento basado en relaciones, como por ejemplo las restricciones en la topología geométrica entre varios puntos (puntos vecinos).

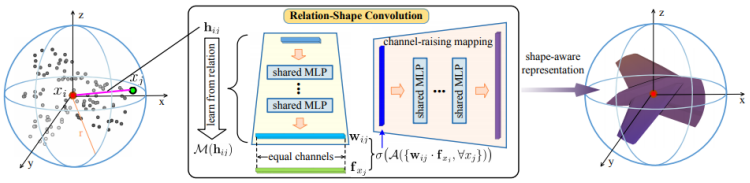


Figura 2.25: Estructura de RSNet [27].

La red está compuestas por varias capas MLP (perceptrón multicapa) que se encargan de aprender un mapeo en base a una relación geométrica predefinida por los vectores. De esta forma la representación convolucional puede razonar el diseño de los puntos, resultando en una discriminación basada en el conocimiento forma. Los autores nos mencionan además que este modelo tiene ciertas propiedades que le dan ventaja frente a otras redes contemporáneas:

- Invariante ante las permutaciones ya que en la función de mapeo interno tanto las redes MLP como las relaciones de bajo nivel son invariantes ante el orden de sus entradas.

- Robustez ante transformaciones rígidas gracias al codificado de alto nivel.

- Interacción entre los puntos ya que al relacionarse entre ellos se crea una región geométrica en el espacio con un significado.

- Compartimiento de pesos. Esta es la clave para poder compartir la función de aprendizaje para diferentes nubes de puntos irregulares.

Los principales usos de esta red son en tareas de segmentación y de clasificación de nubes de puntos.

2.2.4 Minkowski Engine

Minkowski Convolutional Neural Network [28] es una red creada por Christopher Choy, JunYoung Gwak y Silvio Savarese en 2019 con una propuesta innovadora para la extracción de datos de nubes de puntos. Los autores proponen una red neuronal convolucional 4-dimensional, capaz de procesar datos directamente desde un vídeo 3D.

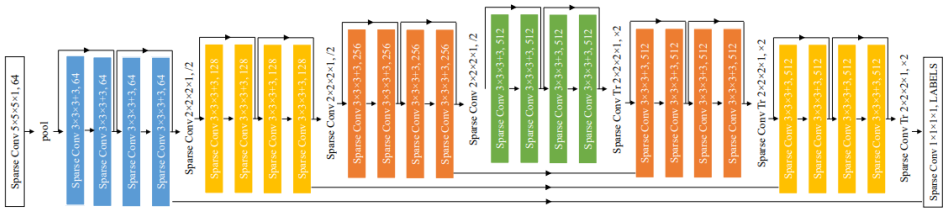


Figura 2.26: Estructura de la red Minkowski engine [27].

Por lo que dicen sus autores se basaron en la arquitectura de la red ResNet18 [29], sobre la cual implementaron un conjunto de capas para conseguir trabajar en un espacio 4-dimensional:

1. **Sparse Tensor Quantization**: Estas primeras capas se encargan de convertir los datos de entrada en tensores de tipo *sparse*.

2. **Generalized Sparse Convolution**: La siguiente etapa genera las coordenadas de las salidas dadas las de las entradas. Después los mapea para poder así identificar qué entradas afectan a qué salidas, y poder pasarlos posteriormente por un *kernel* (núcleo). Por último, dadas las coordenadas de las entradas y salidas, el núcleo y los pesos del mismo, calcula la *convolución escasa generalizada* de forma iterativa.

3. **Max Pooling**: las salidas son llevadas a una capa maxpooling customizada para poder trabajar con datos, ya que en estos al ser tensores de tipo *sparse*, el número de características de entrada varía con cada output.

4. **Global/Average Pooling, Sum Pooling**: Estas capas sirven para calcular la media de características de entrada para cada coordenada de salida.

5. **Non-spatial Functions**: Para acabar podemos usar funciones de activación que no requieren información espacial, como es el caso de la ReLU.

Para terminar, la red puede hacer uso tanto de un kernel híbrido como de uno tipo teseracto. Ambos están pensados para poder trabajar en las 4 dimensiones que requiere la red, pero en el caso del de tipo teseracto requiere de un costo computacional mayor.

Capítulo 3

Estudio del framework

Lo poco que he aprendido carece de valor, comparado
con lo que ignoro y no desespero en aprender.

Descartes

En este capítulo se estudiarán las posibilidades que brinda el framework Pytorch Points 3D [10]. Para ello se analizará la información disponible tanto en el repositorio de Github como en la documentación oficial. Adicionalmente se trabajará con una serie de notebooks, los cuales se adaptarán para que funcionen en Google Colab y se modificarán para comprender su funcionamiento y poder adaptarlos para resolver nuevos problemas.

3.1 Apuntes previos

3.1.1 Estructura del framework

Antes de comenzar a trabajar con el framework se debe entender su estructura, ya que se requerirá de modificar varios ficheros en el futuro. El framework consta de un directorio principal subdividido en las siguientes carpetas:

- **Applications**: En esta carpeta se encuentran los módulos específicos usados en los ejemplos proporcionados por el autor. Principalmente se hallan los scripts necesarios para cargar las redes.
- **Core**: En esta carpeta se encuentran los ficheros necesarios para realizar la mayoría de tareas. Por ejemplo se pueden encontrar la definición de las transformaciones disponibles para ser efectuadas sobre los datos o la definición de los diferentes tipos de pérdidas.
- **Datasets**: En esta carpeta se encuentran los ficheros relacionados con las bases de datos. En ellos se pueden encontrar diversas subcarpetas, las cuales llevan el nombre de la tarea a realizar y contienen los scripts preparados para cargar diferentes bases de datos con las configuraciones óptimas para realizarlas. Por ejemplo, dentro de la carpeta *classification* se encuentra un script

para realizar la descarga de la base de datos *Modelnet*, con el cual se descargan y preparan los datos para poder trabajar con ellos fácilmente en tareas de clasificación.

- **Metrics**: En esta carpeta encontramos principalmente las definiciones de los *trackers* que usaremos para visualizar los resultados de los entrenamientos y el código relacionado con las métricas. La función principal de estos ficheros es llevar un registro de la fase de entrenamiento, de forma que se puedan comprobar los resultados.
- **Models**: En esta carpeta se encuentran los diferentes módulos integrados en el framework, los cuales se explicaron en el capítulo 2.2. Estas definiciones son genéricas y no están preparadas para realizar una tarea concreta.
- **Modules**: Esta carpeta esta relacionada con la de *models* ya que en ella se encuentran los mismos módulos pero preparados para poder trabajar con ellos en alguna tarea determinada.
- **Utils**: En esta carpeta se encuentran diversas funciones y librerías de utilidad. Básicamente recoge funciones que son comunes a varios scripts, tales como la función para descargar datos o algunas funciones matemáticas.
- **Visualization**: En esta carpeta encontramos el código necesario para poder visualizar las nubes de puntos en ventanas.

Comprender la organización del framework es fundamental para poder realizar las modificaciones necesarias que permitan aplicar el mismo en la resolución de nuevos problemas.

3.1.2 Configuraciones previas en Google Colab

Este apartado se llevó a cabo desde Google Colab, un servicio de Google de alojamiento de máquinas virtuales basado en los notebooks de Jupyter que permite trabajar con GPU.

Como Pytorch Points 3D es una librería relativamente nueva se debe descargar manualmente. También hay que descargar los módulos *xvfb x11-utils* y *pyvista* para poder visualizar los resultados, quedando un bloque de descarga de la siguiente forma:

```
#Aquí descargamos Pytorch y todos los módulos necesarios. Si no funciona !pip install torch_points3d
!pip install --upgrade git+https://github.com/nicolas-chaulet/torch-points3d.git

'''Hay que eliminar las versiones existentes de torch-scatter, torch-sparse y torch-cluster ya que Google Colab los tiene en una
versión de CUDA posterior a la necesaria, y si no los eliminamos usa los que vienen por defecto'''

!pip uninstall torch-scatter torch-sparse torch-cluster -y
!rm -rf ~/.cache/pip
!pip install --no-cache-dir torch-scatter torch-sparse torch-cluster -f https://pytorch-geometric.com/whl/torch-1.9.0+cu102.html

#Y aquí un par de librerías necesarias:
!apt-get install -y xvfb x11-utils
!pip install pyvista
```

Figura 3.1: Bloque de descarga Pytorch Points 3D.

Para evitar problemas lo recomendable es tener en cuenta la versión de la GPU disponible en ese momento y ajustar los módulos *torch-scatter*, *torch-sparse* y *torch-cluster* para que no haya

incompatibilidad de versiones. Para comprobar las versiones de estos componentes se pueden usar los siguientes comandos, donde además se pueden verificar las versiones que se usaron a lo largo del trabajo:

```
import sys
import torch

#Comprobamos que la versión de Python sea como mínimo la 3.6
print("Python:\n", sys.version)
#La de CUDA al menos la 10
print("\nCuda: ", torch.version.cuda)
! nvcc --version
#Y la de Pytorch al menos la 1.7
print("\nPytorch:\n",torch.__version__)

#También mostramos la versión del pip
print("\nPip:")
! pip --version
```

```
Python:
 3.7.10 (default, May  3 2021, 02:48:31)
[GCC 7.5.0]

Cuda:  10.2
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2020 NVIDIA Corporation
Built on Wed_Jul_22_19:09:09_PDT_2020
Cuda compilation tools, release 11.0, V11.0.221
Build cuda_11.0_bu.TC445_37.28845127_0

Pytorch:
 1.9.0+cu102

Pip:
pip 19.3.1 from /usr/local/lib/python3.7/dist-packages/pip (python 3.7)
```

(a) Verificación de la versión del GPU.

```
!pip list | grep torch
```

```
pytorch-metric-learning       0.9.99
torch                         1.9.0+cu102
torch-cluster                 1.5.9
torch-geometric               1.7.1
torch-points-kernels          0.7.0
torch-points3d                0.2.0
torch-scatter                 2.0.7
torch-sparse                  0.6.10
torchfile                     0.1.0
torchnet                      0.0.4
torchsummary                  1.5.1
torchtext                     0.10.0
torchvision                   0.10.0+cu102
```

(b) Verificación de la versión de los módulos.

Figura 3.2: Verificación de versiones.

3.2 Estudio del framework en base a los notebooks de ejemplo

Para comprender las posibilidades que ofrece este framework, se estudiará su funcionamiento basándonos en dos notebooks de ejemplo proporcionados por el creador a través de su Github [10]. El primero de ellos trata de un problema de clasificación usando la base de datos ModelNet [23], y para su análisis la arquitectura Relation-Shape Net [27]. El segundo es un problema de segmentación, usando datos de Shapenet [14] y una arquitectura KPConv [24].

3.2.1 Base de datos

El propio framework trae soporte para trabajar con varias bases de datos mostradas en el capítulo 2.2, por lo que para trabajar con cualquiera de ellas lo que se tiene que hacer es definir una serie de parámetros configurables, como por ejemplo las transformaciones, y lanzar una sentencia con la siguiente forma:

```
from torch_points3d.datasets.Y.X import Z
dataset = Z(parámetros)
```

Donde Y corresponde con la carpeta referente a la tarea a realizar y X con la base de datos (como se explicó en la sección 3.1.1). La Z representa el nombre de la clase que se encargará de descargar y organizar la base de datos. Los parámetros son un conjunto de instrucciones que se pueden definir a

conveniencia. En ellos se especifica por ejemplo las transformaciones que sufrirán los datos. Existen varias formas de definir dichos parámetros, pero la que permite una mayor flexibilidad es hacerlo a través de un fichero *.yaml*.

Los scripts preparados para la carga de datos se encuentran en la carpeta *datasets* y pueden ser modificados por los usuarios. Por ejemplo, se puede cambiar la estructura final que tendrán los datos tal y como se verá en futuros apartados. Por ello conviene realizar un par de apuntes relacionados con estos *scripts*:

- Por lo general estos ficheros cuentan con una función primaria que se encarga de realizar el cargado de datos. Esta clase es la que definimos antes como *Z* y es la que se encarga de realizar las separaciones entre grupos de datos (entrenamiento, testeo y validación) y realiza las transformaciones necesarias para cada set.
- Para cargar cada uno de los sets primero se crea una carpeta *process* en la que se guardarán los datos cargados y procesados. Si esta carpeta ya existe directamente se leen los datos de ella usando *torch.load*.
- Si la carpeta no existe se llevan a cabo una serie de pasos por los que los datos son descargados, se les aplica las transformaciones (en caso de haberlas) y son guardados en sus respectivos directorios para posteriormente ser cargados usando *torch.load*.
- Para la fase de descarga se usa la función *download_url()* la cual está definida en la carpeta *utils* dentro del propio framework, explicada en la sección 3.1.1 y *extract_zip()* de la librería *Torch_geometric* [11].
- Para la fase de procesado se leen los datos descargados usando funciones como *open()*, *read()* o *split()*, y luego se almacenan en un objeto tipo *Data*, procedente de la librería *Torch_geometric* [11].

Conocer cómo organiza el framework las bases de datos será de utilidad en el futuro ya que se requerirá de realizar modificaciones sobre ellas. Además este conocimiento da pie a la posibilidad de trabajar con otras bases de datos no presentes en el framework.

3.2.1.1 Base de datos ModelNet (clasificación)

Para el primer ejemplo se cargará la base de datos ModelNet [23]. Esta base de datos permite trabajar con la versión completa de 40 objetos o con una simplificada de sólo 10. Por las limitaciones de Google Colab se usará la versión simplificada. Para su descarga se debe modificar la sentencia mostrada anteriormente de la siguiente manera:

```
from torch_points3d.datasets.classification.modelnet import ModelNetDataset
dataset = ModelNetDataset(params)
```

Como se mencionó anteriormente la variable *params* contiene la definición de los parámetros necesarios para poder realizar la descarga de los datos. Para su creación el autor propone diversas

alternativas, pero la opción que brinda mayor libertad, y por tanto es la que se usará a lo largo del trabajo es crearla a partir de un archivo *.yaml*, usando una variable tipo *OmegaConf* [30]. En este caso el *.yaml* sugerido queda de la siguiente forma:

```
yaml_config = """
task: classification
class: modelnet.ModelNetDataset
name: modelnet
dataroot: %s
number: %s
normal: %r
split: 0.2,0.8
first_subsampling: 0.02

pre_transforms:
    - transform: NormalizeScale
    - transform: GridSampling3D
      params:
        size: ${first_subsampling}

    - transform: FixedPoints
      lparams: [2048]
    - transform: AddFeatsByKeys
      params:
        feat_names: [norm]
        list_add_to_x: [%r]
        delete_feats: [True]

train_transforms:
    - transform: RandomNoise
    - transform: RandomRotate
      params:
        degrees: 180
        axis: 2

""" % (os.path.join(DIR, "data"),MODELNET_VERSION, USE_NORMAL, USE_NORMAL)

from omegaconf import OmegaConf
params = OmegaConf.create(yaml_config)
```

Figura 3.3: Configuración de los parámetros de ModelNet.

Como se puede ver en la figura 3.3 lo primero que se hace es especificar la base de datos a cargar, así como la tarea que se desarrollará y cómo dividir los distintos sets (con la variable split). En este caso se designarán un 80 % de los datos al set de entrenamiento, mientras que el 20 % restante irá al de testeo. Seguidamente se especifican las transformaciones que se deseen realizar sobre los datos. En esta ocasión se realizarán diferentes transformaciones para cada conjunto de datos:

- **Pre transforms**: Estas son las transformaciones que se aplicarán a todos los conjuntos de datos. Se especifican cuatro transformaciones:
  - NormalizeScale: Realiza una transformación donde centra y normaliza las posiciones de los nodos en un intervalo (-1,1).
  - GridSampling3D: Agrupa los puntos en vóxeles de tamaño *first subsampling*, variable definida anteriormente.
  - FixedPoints: Asegura que todos los objetos tengan el mismo número de puntos (2048).

  - AddFeatsByKeys: Añade una serie de atributos a la lista. En este caso sería un vector normal.

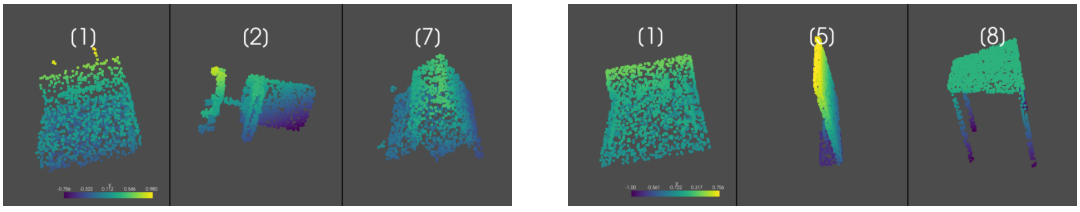
- **Train transforms**: Estas transformaciones se llevarán a cabo exclusivamente sobre el conjunto de datos de entrenamiento. Con la intención de mejorar la capacidad de aprendizaje de la red, este conjunto de datos es sometido a un par de transformaciones extras, *RandomNoise* y *RandomRotate* para aplicar ruido y rotarlas de manera aleatoria. Con esto se consigue que la red pueda encontrar características más generales. Por ejemplo, la base de datos nos proporciona los objetos siempre con la misma orientación, y al rotarlos se intenta mitigar que esto tenga efecto.

Finalmente se adjunta la dirección donde queramos guardar los datos descargados, se selecciona la versión (completa o simplificada) y se especifica el uso del vector normal. De esta forma se logra cargar y separar fácilmente los datos que se usarán a lo largo del ejercicio. Finalmente se puede comprobar que todo ha ido correctamente mostrando la variable *dataset*:

```
Dataset: ModelNetDataset
train_pre_batch_collate_transform = None
val_pre_batch_collate_transform = None
test_pre_batch_collate_transform = None
pre_transform = Compose([
    NormalizeScale(),
    GridSampling3D(grid_size=0.02, quantize_coords=False, mode=mean),
    FixedPoints(2048, replace=True),
    AddFeatsByKeys(norm=True),
])
test_transform = None
train_transform = Compose([
    RandomNoise(sigma=0.01, clip=0.05),
    RandomRotate((-180, 180), axis=2),
])
val_transform = None
inference_transform = Compose([
    NormalizeScale(),
    GridSampling3D(grid_size=0.02, quantize_coords=False, mode=mean),
    FixedPoints(2048, replace=True),
    AddFeatsByKeys(norm=True),
])
Size of train_dataset = 3991
Size of test_dataset = 908
Size of val_dataset = 0
Batch size = None
```

Figura 3.4: Datos cargados de ModelNet con las especificaciones presentes en el yaml.

Como se puede apreciar en la figura 3.4 todas las transformaciones se han llevado a cabo correctamente, y se han generado dos conjuntos diferentes, uno con 3991 muestras que se usará para el entrenamiento de la red (*train dataset*) y otro con 908 que será de utilidad para verificar que el entrenamiento se lleve a cabo correctamente (*test dataset*) A continuación se pueden ver varios ejemplos extraídos de ambos conjuntos donde se pueden apreciar las transformaciones realizadas sobre el set de entrenamiento:



(a) Muestras del set de entrenamiento.

(b) Muestras del set de testeo.

Figura 3.5: Muestras extraídas de ModelNet.

3.2.1.2 Base de datos ShapeNet (segmentación)

El segundo notebook usa nubes de puntos procedentes de Shapenet [14], base de datos que también forma parte del framework, por lo que para cargar los datos basta con modificar el código de una forma parecida:

```
from torch_points3d.datasets.segmentation import ShapeNetDataset
dataset = ShapeNetDataset(params)
```

Donde de para definir la variable *params* se crea un nuevo *yaml* con las especificaciones requeridas para el ejercicio:

```
#Configure the dataset {run: "auto"}
CATEGORY = "Airplane" #@param ["Airplane", "Car", "Table", "Bag", "All",
USE_NORMALS = True #@param {type:"boolean"}

shapenet_yaml = """
class: shapenet.ShapeNetDataset              #Base de datos utilizada
task: segmentation                           #Tarea a realizar
dataroot: %s
normal: %r                                   # Use normal vectors as fea
first_subsampling: 0.02                      # Grid size of the input da
pre_transforms:                              # Offline transforms, done
    - transform: NormalizeScale              # Realiza una transformaci
    - transform: GridSampling3D              # Agrupa los puntos en vóxe
      params:
        size: ${first_subsampling}
    - transform: AddOnes                     #Añade el tensor de 1 a dat
    - transform: AddFeatsByKeys              #Añade los siguientes atrib
      params:
        list_add_to_x: [True]                #Controla si la característ
        feat_names: ["ones"]                 #La lista "ones" de caracte
        delete_feats: [True]                 #Habilitamos que se puedan
train_transforms:                            # Data augmentation pipelin
    - transform: RandomNoise                 #Aplica ruido a los datos c
      params:
        sigma: 0.01
        clip: 0.05
    - transform: RandomScaleAnisotropic      #Escala los nodos en un val
      params:
        scales: [0.9,1.1]
""" % (os.path.join(DIR,"data"), USE_NORMALS)

from omegaconf import OmegaConf
params = OmegaConf.create(shapenet_yaml)
if CATEGORY != "All":
    params.category = CATEGORY
```

Figura 3.6: Configuración de los parámetros de ShapeNet.

La estructura general es muy parecida a la que se explicó anteriormente en la figura 3.3, con la diferencia de que en este caso al ser una tarea de segmentación se pueden elegir los modelos a descargar. Por ejemplo, se permite descargar solo los modelos de los aviones sin necesidad de descargar el resto de objetos. El resto de la estructura es prácticamente idéntica, solo añadiendo un par de transformaciones nuevas:

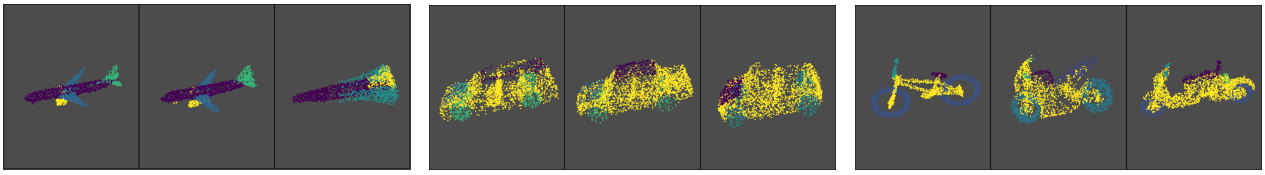
- **AddOnes**: Añade un tensor de 1 a *data*. Esto es necesario para cumplir los requisitos de entrada de KPConv.
- **RandomScaleAnisotropic**: Escala los nodos en un valor aleatorio dentro del intervalo dado (0,9-1,1)

En este caso se realiza además una transformación *RandomScale* al set de entrenamiento ya que al tener que diferenciar las partes de las que están compuestas los objetos, estos escalados aleatorios ayudaran a generalizar las características. De nuevo si se observa la variable en la que se guardan los datos se puede comprobar que todo se ha descargado correctamente:

```
Dataset: ShapeNetDataset
train_pre_batch_collate_transform = None
val_pre_batch_collate_transform = None
test_pre_batch_collate_transform = None
pre_transform = Compose([
    NormalizeScale(),
    GridSampling3D(grid_size=0.02, quantize_coords=False, mode=mean),
    AddOnes(),
    AddFeatsByKeys(ones=True),
])
test_transform = None
train_transform = Compose([
    RandomNoise(sigma=0.01, clip=0.05),
    RandomScaleAnisotropic([0.9, 1.1]),
])
val_transform = None
inference_transform = Compose([
    NormalizeScale(),
    GridSampling3D(grid_size=0.02, quantize_coords=False, mode=mean),
    AddOnes(),
    AddFeatsByKeys(ones=True),
])
Size of train_dataset = 1958
Size of test_dataset = 341
Size of val_dataset = 391
Batch size = None
```

Figura 3.7: Dataset cargado de ShapeNet.

También se verificará que la descarga ha sido correcta visualizando algunos de los modelos de los diferentes sets:



(a) Muestras de modelos de aviones. (b) Muestras de modelos de coches. (c) Muestras de modelos de motos.

Figura 3.8: Muestras extraídas de ShapeNet.

En los ejemplos expuestos en la figura 3.8 se puede apreciar la forma de diferenciar los objetos. Cada modelo tiene partes coloreadas de diferentes colores, y cada uno de estos colores representa una parte que hay que clasificar. Por ejemplo, en el caso del avión se divide en cuatro partes (alas, turbinas, fuselaje y estabilizadores).

3.2.2 Organización de los datos

Una vez se ha completado la descarga de los datos, estos se han de organizar para poder trabajar con ellos. Para ello, como las bases de datos forman parte del framework, esta vienen preparadas con un módulo *create_dataloader*. Este se encarga de empaquetar los datos para poder trabajar con ellos en forma de *batch*. Este módulo requiere de las siguientes entradas:

- **Model:** Arquitectura que se va a utilizar.
- **Batch size:** Variable que especifica el tamaño de cada conjunto.
- **Shuffle:** Si esta variable es *true* mezcla los datos para evitar que aparezcan en orden.
- **Num workers:** Si esta variable tiene un valor positivo activa el procesado de datos múltiple con el número de trabajadores especificado. Si por ejemplo se define con un valor de 4, se crearán cuatro hilos que trabajaran en paralelo para cargar los datos.
- **Precompute multiscale:** Si tiene algún valor los datos serán escalados. Si es *false* no pasará nada.

3.2.2.1 Organización de los datos de ModelNet (clasificación)

En este dataset se van a empaquetar los datos en batches de tamaño 12 para mejorar la eficacia del entrenamiento, y se especificará un número de trabajadores de 4. Como la base de datos ha sido cargando haciendo uso del framework *Pytorch Points 3D* [10] la variable cuenta con el módulo *create_dataloader* mencionado anteriormente. Para cargarlo lanzamos la siguiente sentencia:

```
NUM_WORKERS = 4
BATCH_SIZE = 12

dataset.create_dataloaders(
    model,
    batch_size=BATCH_SIZE,
    shuffle=True,
    num_workers=NUM_WORKERS,
    precompute_multi_scale=False
)
```

Figura 3.9: Agrupamiento de los datos de Modelnet.

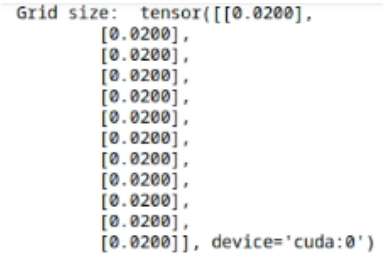
De esta forma los datos quedan preparados para poder trabajar con ellos, pero antes de continuar se estudiará más a fondo la estructura de los mismos. Lo primero que se observa es la estructura de los *batches*:

```
SimpleBatch(grid_size=[12, 1], pos=[12, 2048, 3], x=[12, 2048, 3], y=[12, 1])
```

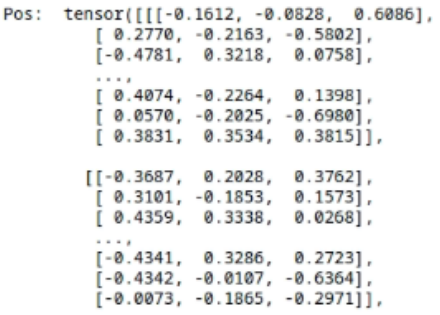
Figura 3.10: Estructura de un batch para hacer la clasificación.

Como se puede ver en la figura 3.10 los datos se han almacenado en variables tipo *SimpleBatch*, y estas serán las entradas de la red en el futuro. *SimpleBatch* es un tipo de ensamblado *dense batch*,

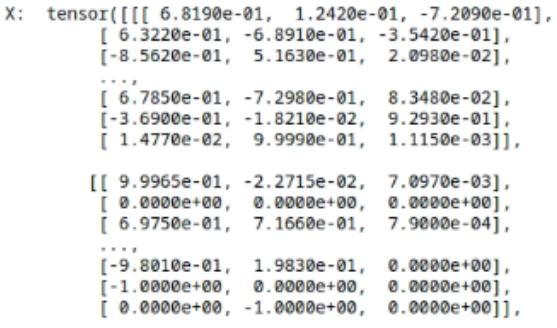
que significa que los datos han sido muestreados para tener el mismo tamaño y serán cotejados en la nueva dimensión del batch. Lo primero que se aprecia es que todos los campos cuentan tensores de tamaño [12,X], lo cual tiene sentido ya que se ha configurado un tamaño de batch de 12 muestras por paquete. Estos campos contienen los siguientes datos:



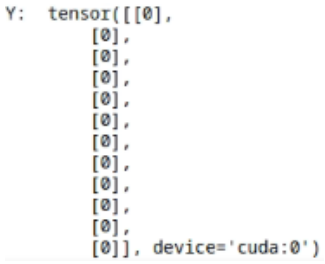
(a) Parámetro grid size.



(b) Parámetro pos.



(c) Parámetro x.



(d) Parámetro y.

Figura 3.11: Parámetros del Simplebatch de ModelNet.

- **Grid size**: Parámetro que almacena el tamaño del *grid*. Es un tensor [12, 1] ya que se cuenta con 12 muestras en cada *batch* y cada una de ellas tiene un valor de *grid*.
- **Pos**: Parámetro que almacena los puntos que componen el modelo. Es un tensor [12, 2048, 3] ya que para cada muestra hay 2048 puntos (como se especificó en el *yaml* de la figura 3.3) y a su vez cada punto está compuesto por tres coordenadas.
- **X**: Parámetro que almacena el vector normal a cada punto. Como hay 2048 puntos existen 2048 vectores normales.
- **Y**: Parámetro que almacena la salida ideal (*groundtruth*) de cada muestra. Es un tensor [12, 1] ya que cada muestra tiene un valor de salida ideal.

3.2.2.2 Organización de los datos de Shapenet (segmentación)

Para los datos de shapenet se aplica el mismo criterio, solo cambiando el tamaño de *batch* a 16. Al estar definida la clase de distinta forma los datos tienen una estructura diferente a la del caso anterior, ya que la tarea que van a realizar es diferente:

```
MultiScaleBatch(batch=[34447], category=[34447], grid_size=[16], id_scan=[16], multiscale=[10],
 origin_id=[34447], pos=[34447, 3], ptr=[17], upsample=[4], x=[34447, 4], y=[34447])
```

Figura 3.12: Estructura de los datos cargados de ShapeNet.

En el caso anterior (sección 3.2.2.1) al tratarse de una tarea de clasificación los diferentes objetos se almacenan en batches separados donde cada paquete engloba a los puntos de la muestra, y son estos puntos los que cuentan con características (en el anterior ejemplo era el modelo al que se le asociaban una serie de características, en este las características son asociadas a cada punto individualmente). Se debe tener en cuenta esta consideración para poder entender su estructura, la cual se explica a continuación:

- **x [N,4]**: Vector de características. En este caso las características son el vector normal y un 1 (añadido con la transformación *AddOnes* explicada en la figura 3.7) por lo que el tamaño del vector es *N (número de puntos) por 4 (vector normal + vector 1)*.
- **y [N]**: Vector de resultados ideales que identifica la parte del modelo que representa cada punto. Cada punto tiene una salida ideal por lo que el tamaño del vector es *N (número de puntos)*.
- **pos [N,3]**: Vector que contiene las posiciones de todos los puntos. Como se trabaja en un espacio tridimensional el tamaño es *N (número de puntos) por 3 (número de dimensiones)*.
- **multiscale**: Vector que almacena las distintas capas del objeto. El tamaño representa el número de capas, y viene determinado por la base de datos.
- **upsample**: Vector que contiene las identificaciones de los puntos vecinos. Viene determinado por la base de datos.
- **batch [N]**: Vector que indica el *batch*. Hay un batch por cada punto, por lo que es un vector de tamaño *N (número de puntos)*.
- **category [N]**: Vector que representa el tipo de objeto de la muestra. Como todos los puntos de una muestra representan el mismo objeto este vector tiene tamaño *N (número de puntos)* pero todas las celdas tienen el mismo valor.
- **grid_size [B]**: Vector que representa el tamaño del batch. Como en el *dataloader* se han dividido en paquetes de tamaño 16 este vector es de tamaño *B (tamaño del batch)* y las 16 celdas contienen el mismo valor.
- **id_scan [B]**: Vector que contiene la identificación del punto dentro del modelo de forma ordenada. Como los paquetes son de 16 puntos el tamaño es *B (tamaño del batch)*.
- **original_id [N]**: Vector que contiene la identificación de todos los puntos del modelo. Estos aparecen de forma desordenada. Como engloba todos los puntos del modelo es de tamaño *N (número de puntos)*.
- **ptr (Production Resources and Tools)**: Recursos y herramientas que se generan al hacer el batch.

Como se explicó en el capítulo 2.2 esta red entrena en base a un núcleo de puntos vecinos. En la figura 3.13 se puede apreciar como cada punto de la ventana de la derecha tiene asociado una serie de puntos cercanos mostrados en la ventana de la izquierda. Esta red estudia este tipo de relaciones. Para ello además hace uso de modelos con menos capas. Como se vio anteriormente los datos tienen un parámetro que define el número de capas, lo que permite variar el número de estas tal y como se muestra en la figura 3.14.

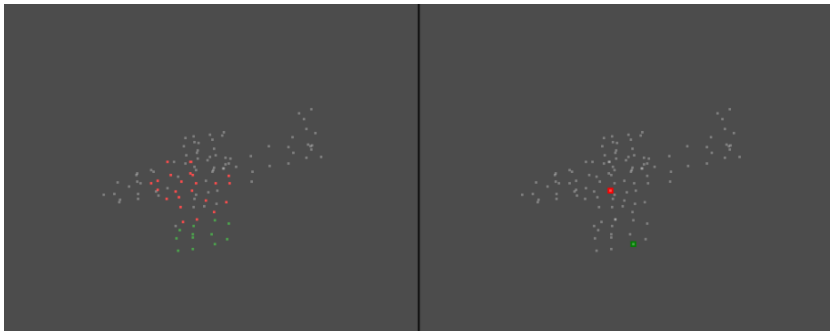


Figura 3.13: Puntos vecinos.

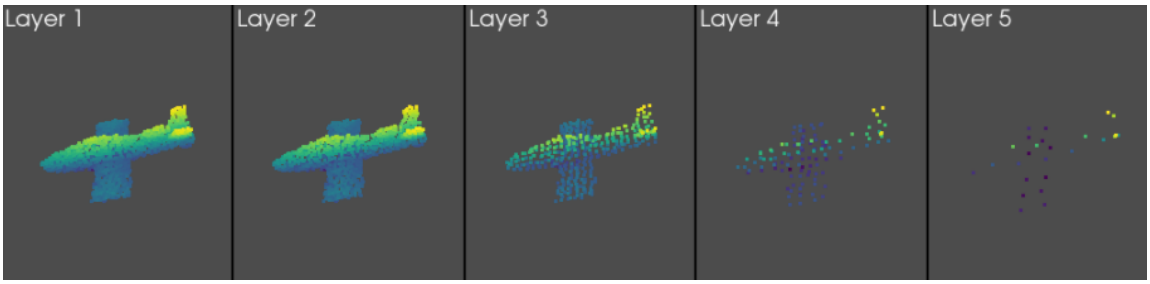


Figura 3.14: Diferentes capas de la nube de puntos de un modelo de avión.

3.2.3 Creación del modelo

Al igual que con los datos, si la arquitectura se encuentra dentro del framework para cargarla solo es necesario importarla desde el lugar donde se encuentra. Las arquitecturas principales que requieren ambos ejemplos pueden ser encontradas en la carpeta *applications*, mencionada en la sección 3.1.1, por lo que para importarlas la sentencia tendrá la siguiente estructura:

```
from torch_points3d.applications.Y   import    X
```

Donde Y corresponde con la carpeta en la que se guarda la arquitectura y X con la clase que la define. Al igual que en el caso de las bases de datos del capítulo 3.2.1, el framework incluye una serie de ficheros preparados para trabajar con las arquitecturas mencionadas en el capítulo 2.2, los cuales almacena en la carpeta *modules*. En ellos se encuentran todas las funciones necesarias para cargar la arquitectura y los ficheros necesarios para realizar el cálculo de las pérdidas. Adicionalmente, en la carpeta *applications* se pueden encontrar ficheros en los que se preparan estas arquitecturas para ser usadas en los ejemplos. Estos scripts únicamente se encargan de cargar la arquitectura y de definir su funcionamiento.

3.2.3.1 Creación del modelo Relation-Shape Net (clasificación)

Para el primer ejemplo se usa una arquitectura Relation-Shape Net [27] en modo clasificación. Ya que RSNet forma parte del framework para importarla basta con modificar la sentencia de la siguiente forma:

```
from torch_points3d.applications.rsconv import RSConv
```

Lo siguiente será crear una clase que defina el funcionamiento de la red. Para ello se usa RSNet como arquitectura principal y se añade una capa *softmax* al final, la cual se encargará de realizar la clasificación:

```
class RSConvClassifier(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.encoder = RSConv("encoder", input_nc= 3 * USE_NORMAL,output_nc = int(MODELNET_VERSION), num_layers=4)
        self.log_softmax = torch.nn.LogSoftmax(dim=-1)

    @property
    def conv_type(self):
        """ This is needed by the dataset to infer which batch collate should be used"""
        return self.encoder.conv_type

    def get_output(self):
        """ This is needed by the tracker to get access to the ouputs of the network"""
        return self.output

    def get_labels(self):
        """ Needed by the tracker in order to access ground truth labels"""
        return self.labels

    def get_current_losses(self):
        """ Entry point for the tracker to grab the loss """
        return {"loss_class": float(self.loss_class)}

    def forward(self, data):
        # Set labels for the tracker
        self.labels = data.y.squeeze()

        # Forward through the network
        data_out = self.encoder(data)
        self.output = self.log_softmax(data_out.x.squeeze())

        # Set loss for the backward pass
        self.loss_class = torch.nn.functional.nll_loss(self.output, self.labels)

    def forward_V2(self, data):
        #Función de forward pero sin el cálculo de pérdidas para sacar los resultados
        # Forward through the network
        data_out = self.encoder(data)
        self.output = self.log_softmax(data_out.x.squeeze())

        return self.output

    def backward(self):
         self.loss_class.backward()
```

Figura 3.15: Estructura de la red de clasificación.

Para facilitar la creación de la clase esta es hereda de *torch.nn.Module*, pudiendo aprovecharse así de funciones ya creadas que facilitan su implementación, como por ejemplo el uso de la función *softmax*. En esta clase lo primero que se hace es crear el *__init()__* con las arquitecturas que serán usadas:

- **RSConv**: Se trata de la arquitectura Relation-Shape Net [27], la cual se define en modo *encoder* configurándola así para tareas de clasificación. Se fijan tres canales de entrada (uno por cada dimensión que se va a analizar) y el número de canales de salida corresponderá con la cantidad de modelos a clasificar. En este caso como se usa la versión simplificada la salida es de tamaño 10. Por último se define el número de capas a 4.

- **Log_softmax**: Para poder clasificar correctamente los objetos se usa una capa final que implementa la función *softmax*.

Lo siguiente que se debe definir es cómo pasar los datos a través de la red. Para ello se crea la función *forward*. En esta función se puede ver que los datos primero pasan por el *encoder*, el cual se trata del modelo RSNet, y seguidamente por la función softmax, la cual genera el vector de salida. Además de esto, en la propia función las etiquetas son procesadas por una función "squeeze()". Esta asegura que las etiquetas sean de tamaño unidimensional. Por último se realiza el cálculo de las pérdidas. Puesto que no siempre se contará con las etiquetas se ha creado una segunda función *forward_V2*, la cual es idéntica a la primera con la excepción de que esta no realiza el cálculo de pérdidas, haciendo posible su funcionamiento sin necesidad de etiquetas. El resto de funciones son requeridas para los procesos de seguimiento.

3.2.3.2 Creación del modelo KPConv (segmentación)

En el segundo ejemplo se usa una arquitectura KPConv [24]. Puesto que en este proyecto se pueden segmentar distintos objetos, se apoyará esta estructura con una capa de clasificación. Dicha capa está diseñada para poder trabajar con todos los modelos disponibles, de forma que la salida final será un vector de tamaño dependiente de la muestra, donde cada celda corresponde con una parte de un objeto. Así, si por ejemplo estamos segmentando un avión, las celdas que nos interesan serán las cuatro primeras, mientras que si hablamos de un coche serán las celdas de la 8 a la 11.

El clasificador está diseñado tal y como se muestra en la figura 3.16.

```python
from torch_points3d.core.common_modules import MLP, UnaryConv

class MultiHeadClassifier(torch.nn.Module):

    def __init__(self, in_features, cat_to_seg, dropout_proba=0.5, bn_momentum=0.1):
        super().__init__()
        self._cat_to_seg = {}
        self._num_categories = len(cat_to_seg)
        self._max_seg_count = 0
        self._max_seg = 0
        self._shifts = torch.zeros((self._num_categories,), dtype=torch.long)
        for i, seg in enumerate(cat_to_seg.values()):
            self._max_seg_count = max(self._max_seg_count, len(seg))
            self._max_seg = max(self._max_seg, max(seg))
            self._shifts[i] = min(seg)
            self._cat_to_seg[i] = seg

        self.channel_rasing = MLP(
            [in_features, self._num_categories * in_features], bn_momentum=bn_momentum, bias=False
        )
        if dropout_proba:
            self.channel_rasing.add_module("Dropout", torch.nn.Dropout(p=dropout_proba))

        self.classifier = UnaryConv((self._num_categories, in_features, self._max_seg_count))
        self._bias = torch.nn.Parameter(torch.zeros(self._max_seg_count,))
```

(a) Init del clasificador.

```python
def forward(self, features, category_labels, **kwargs):

    #print('Entrando en el clasificador')
    assert features.dim() == 2
    self._shifts = self._shifts.to(features.device)
    in_dim = features.shape[-1]
    features = self.channel_rasing(features)
    features = features.reshape((-1, self._num_categories, in_dim))
    features = features.transpose(0, 1)  # [num_categories, num_points, in_dim]
    features = self.classifier(features) + self._bias  # [num_categories, num_points, max_seg]
    ind = category_labels.unsqueeze(-1).repeat(1, 1, features.shape[-1]).long()

    logits = features.gather(0, ind).squeeze(0)
    softmax = torch.nn.functional.log_softmax(logits, dim=-1)

    output = torch.zeros(logits.shape[0], self._max_seg + 1).to(features.device)
    #print('Output 1: ', output)
    cats_in_batch = torch.unique(category_labels)
    #print('Categorías: ', cats_in_batch)
    for cat in cats_in_batch:
        cat_mask = category_labels == cat
        seg_indices = self._cat_to_seg[cat.item()]
        probs = softmax[cat_mask, : len(seg_indices)]
        #print('Probabilidad: ',probs)
        output[cat_mask, seg_indices[0] : seg_indices[-1] + 1] = probs

    return output
```

(b) Función forward del clasificador.

Figura 3.16: Estructura de la capa de clasificación.

```python
from torch_points3d.applications.kpconv import KPConv
class PartSegKPConv(torch.nn.Module):
    def __init__(self, cat_to_seg):
        super().__init__()
        #La arquitectura "unet" es la que se usa para la segmentación. Si quisieramos clasificar habría que cambiarlo a "encoder"
        self.unet = KPConv(
            architecture="unet",
            input_nc=USE_NORMALS * 3,
            num_layers=4,
            in_grid_size=0.02
            )
        self.classifier = MultiHeadClassifier(self.unet.output_nc, cat_to_seg)
    @property
    def conv_type(self):
        """ This is needed by the dataset to infer which batch collate should be used"""
        return self.unet.conv_type

    def get_batch(self):
        return self.batch

    def get_output(self):
        """ This is needed by the tracker to get access to the ouputs of the network"""
        return self.output

    def get_labels(self):
        """ Needed by the tracker in order to access ground truth labels"""
        return self.labels

    def get_current_losses(self):
        """ Entry point for the tracker to grab the loss """
        return {"loss_seg": float(self.loss_seg)}
    def forward(self, data):
        self.labels = data.y
        self.batch = data.batch

        # Forward through unet and classifier
        #Primero lo pasa atraves de KPConv:
        data_features = self.unet(data)
        #Y el resultado lo pasa por el modulo clasificador:
        self.output = self.classifier(data_features.x, data.category)

        # Set loss for the backward pass
        self.loss_seg = torch.nn.functional.nll_loss(self.output, self.labels)
        return self.output

    #Añadido para poder ver el output de la red (o no)
    def forward_V2(self, data):

        #Primero lo pasa atraves de KPConv:
        data_features = self.unet(data)
        print('He pasado por KPConv')
        #Y el resultado lo pasa por el modulo clasificador:
        self.output = self.classifier(data_features.x, data.category)
        print('He pasado por el clasificador')
        return self.output

    def get_spatial_ops(self):
        return self.unet.get_spatial_ops()

    def backward(self):
         self.loss_seg.backward()

model = PartSegKPConv(dataset.class_to_segments)
```

Figura 3.17: Estructura de la red completa basada en una arquitectura KPConv.

En la figura 3.16 se muestra la estructura del clasificador. Las características del mismo se definen en el módulo *_init_()*. Como se puede apreciar este tiene 4 entradas, las **características (in_features)** que serán un vector procedente de la red KPConv, las **categorías (cat_to_seg)** que indica de qué objeto se trata, el **dropout (dropout_proba)** que es un hiperparámetro que se usa a la hora de entrenar para mejorar la eficiencia de la red (cancela aleatoriamente algunos nodos lo que mejora la capacidad de entrenamiento de la red) y el **momento (bn_momentum)**, otro hiperparámetro que va modifican el valor del *learning rate* conforme vaya acercándose más a la solución idónea. Lo interesante dentro de esta sección es que para hacer la clasificación se usa una capa *MLP (MultiLayer Perceptron)* y una capa *UnaryConv*, capa propia del framework Pytorch Points 3D [10] que realiza una convolución 1X1 sobre la nube de puntos. El módulo *forward* es el que realiza la clasificación. Para ello las características pasan primero por la capa MLP y después por la UnaryConv. Por último el resultado es desplazado hasta la posición correspondiente dentro del vector de salida. Una vez definido el clasificador se crea la red completa, mostrada en la figura 3.17.

Al igual que en el anterior notebook las partes más importantes son las funciones _init_() y forward() ya que el resto de funciones solo son necesarias para operaciones de seguimiento. El la función **__init__()** se define la arquitectura KPConv. Esta se especifica en modo segmentación (unet) ya que la arquitectura se puede usar también para clasificar objetos. Se establecen 3 canales de entrada (uno por cada dimensión de los datos), 4 capas y el tamaño del *grid*. En la función *forward()* se define el funcionamiento de la red. Los datos de entrada pasaran primero por la arquitectura KPConv (unet(data)) y la salida es obtenida a través del *classifier()*. Como se puede apreciar, en esta función también se lleva a cabo el cálculo de las pérdidas, por lo que para poder usarla en un entorno más general se crea la función *forward_V2()* que no tiene en cuenta este último cálculo.

3.2.4 Entrenamiento

Una vez que los datos se han cargado y la red ha sido creada el siguiente paso que se debe dar es entrenar la arquitectura en la tarea indicada. En este caso el framework no proporciona ningún método determinado para realizar dicho entrenamiento, pero a la hora de cargar los datos y crear el modelo se ha incorporado un módulo *tracker*, el cual irá guardando la información del entrenamiento para su posterior estudio. Para llevar a cabo el entrenamiento en ambos ejemplos se usa un optimizador *Adam* [31]. Este algoritmo pretende mejorar el método estocástico usado habitualmente para la actualización de los pesos al introducir un *learning rate* adaptativo en función del promedio de los dos primeros momentos (media y varianza).

3.2.4.1 Entrenamiento en el notebook de clasificación

Esta red ha de ser capaz de diferenciar los objetos que formas las nubes de puntos. Para ello se crean dos funciones, las cuales se muestran en la figura 3.18.

```python
def train_epoch(device):
    model.to(device)
    model.train()
    tracker.reset("train")
    train_loader = dataset.train_dataloader
    iter_data_time = time.time()
    with Ctq(train_loader) as tq_train_loader:
        for i, data in enumerate(tq_train_loader):
            t_data = time.time() - iter_data_time
            iter_start_time = time.time()
            optimizer.zero_grad()
            data.to(device)
            model.forward(data)
            model.backward()
            optimizer.step()
            if i % 10 == 0:
                tracker.track(model)

            tq_train_loader.set_postfix(
                **tracker.get_metrics(),
                data_loading=float(t_data),
                iteration=float(time.time() - iter_start_time),
            )
            iter_data_time = time.time()
```

(a) Bucle de entrenamiento.

```python
def test_epoch(device):
    model.to(device)
    model.eval()
    tracker.reset("test")
    test_loader = dataset.test_dataloaders[0]
    iter_data_time = time.time()
    with Ctq(test_loader) as tq_test_loader:
        for i, data in enumerate(tq_test_loader):
            t_data = time.time() - iter_data_time
            iter_start_time = time.time()
            data.to(device)
            model.forward(data)
            tracker.track(model)

            tq_test_loader.set_postfix(
                **tracker.get_metrics(),
                data_loading=float(t_data),
                iteration=float(time.time() - iter_start_time),
            )
            iter_data_time = time.time()
```

(b) Bucle de testeo.

Figura 3.18: Bucles para realizar el entrenamiento.

Ambas funciones serán usadas para realizar el entrenamiento, pero antes las explicaremos individualmente:

- **Train_epoch()**: Este bucle se encargará de actualizar el valor de los pesos. Para ello primero se pone el modelo en modo entrenamiento (*model.train()*) y después se pasan los batches creados anteriormente uno a uno a través de la red con la función *model.forward(data)*, donde se realiza el cálculo de pérdidas. Seguidamente con la función *model.backward()* actualiza el valor de los pesos en base al gradiente del error.

- **Test_epoch()**: Este bucle realiza un testeo de la red cada vez que es llamado. Para ello primero se pone el modelo en forma de testeo (*model.eval()*) y después se hace uso de la función *model.forward(data)* para realizar las predicción y anotar los resultados de las pruebas en el *tracker*. Poniéndolo en modo de testeo nos aseguramos de que no se lleven a cabo algunas funciones como el *dropout* que solo son útiles durante la etapa de entrenamiento.

El optimizador se define de la siguiente forma:

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
```

Dentro del optimizador se introducen los parámetros procedentes del modelo y se determina un valor de *learning rate* α=0.01.

3.2.4.2 Entrenamiento en el notebook de segmentación

En este caso el entrenamiento se concentrará dentro de una clase. Esta clase estará compuesta por varios módulos, los cuales al combinarlos permitirán entrenar la red en nuestra base de datos.

```python
class Trainer:
    def __init__(self,model, dataset, num_epoch = 5, device=torch.device('cuda')):
        self.num_epoch = num_epoch
        self._model = model
        self._dataset=dataset
        self.device = device

    def fit(self):
        self.optimizer = torch.optim.Adam(self._model.parameters(), lr=0.001)
        self.tracker = self._dataset.get_tracker(False, True)

        for i in range(self.num_epoch):
            print("========== EPOCH %i ==========" % i)
            time.sleep(0.5)
            self.train_epoch()
            self.tracker.publish(i)
            self.test_epoch()
            self.tracker.publish(i)
```

Figura 3.19: Módulos init() y fit() de la clase trainer.

En la figura 3.19 podemos observar la creación de la clase encargada del entrenamiento. Dentro de esta, en la parte del método *__init__()* se definen las entradas que tendrá esta clase. Como se pude ver, se requiere de introducir la red a entrenar, los datos preparados y, opcionalmente, se puede configurar el número de ciclos y el entorno de ejecución (en este caso al disponer de GPU lo mejor es hacerlo de esta forma). El método *fit()* será el llamada cuando se quiera realizar el entrenamiento. En él se define el optimizador tipo Adam, y dentro de este se inicializa el *learning rate*. Seguidamente se recorre el bucle realizando las fases de entrenamiento y testeo de forma intercalada el número de iteraciones que se hayan especificado en el método *__init__()*.

```python
    def train_epoch(self):
        self._model.to(self.device)
        self._model.train()
        self.tracker.reset("train")
        train_loader = self._dataset.train_dataloader
        iter_data_time = time.time()
        with tqdm(train_loader) as tq_train_loader:
            for i, data in enumerate(tq_train_loader):
                t_data = time.time() - iter_data_time
                iter_start_time = time.time()
                self.optimizer.zero_grad()
                data.to(self.device)
                self._model.forward(data)
                self._model.backward()
                self.optimizer.step()
                if i % 10 == 0:
                    self.tracker.track(self._model)

                tq_train_loader.set_postfix(
                    **self.tracker.get_metrics(),
                    data_loading=float(t_data),
                    iteration=float(time.time() - iter_start_time),
                )
                iter_data_time = time.time()
```

Figura 3.20: Módulo train de la clase trainer.

En la figura 3.20 se pude observar la función de entrenamiento. Esta se comporta igual que la de la figura 3.18, pasando las muestras hacia la salida, calculando las pérdidas y una vez obtenido el resultado, actualizando los pesos con la función backward().

```
def test_epoch(self):
    self._model.to(self.device)
    self._model.eval()
    self.tracker.reset("test")
    test_loader = self._dataset.test_dataloaders[0]
    iter_data_time = time.time()
    with tqdm(test_loader) as tq_test_loader:
        for i, data in enumerate(tq_test_loader):
            t_data = time.time() - iter_data_time
            iter_start_time = time.time()
            data.to(self.device)
            self._model.forward(data)
            self.tracker.track(self._model)

            tq_test_loader.set_postfix(
                **self.tracker.get_metrics(),
                data_loading=float(t_data),
                iteration=float(time.time() - iter_start_time),
            )
            iter_data_time = time.time()
```

Figura 3.21: Módulo test de la clase trainer.

La función de la figura 3.21 también actúa de la misma forma que la de la figura 3.18. Después de entrenar la red pasa un conjunto de modelos de muestra para comprobar los resultados, y así se puede localizar casos de *overfitting* si se llegan a dar.

3.2.5 Resultados

Finalmente se mostrarán los resultados obtenidos en ambos notebooks. Para ello se llevarán a cabo los entrenamientos, analizando los resultados obtenidos de los mismos, y finalmente mostrando algunas imágenes con las predicciones realizadas por las redes.

3.2.5.1 Resultados del notebook de clasificación

Una vez se ha definido todo lo necesario, el último paso será entrenar la red en la base de datos y visualizar los resultados. Para realizar el entrenamiento se hará uso de las funciones definidas en la sección 3.2.4.1 durante 15 ciclos:

```
EPOCHS = 15
for i in range(EPOCHS):
  print("========== EPOCH %i ==========" % i)
  time.sleep(0.5)
  train_epoch('cuda')
  tracker.publish(i)
  test_epoch('cuda')
  tracker.publish(i)
```

Figura 3.22: Bucle de entrenamiento.

Como se puede apreciar en cada ciclo se entrena la red y después se comprueba su rendimiento. De esta forma se puede evaluar cuan efectivo ha sido el entrenamiento en base a los parámetros definidos, y si sucediese *overfitting* en algún momento podría detectarse. A continuación se exponen las gráficas de acierto y pérdidas obtenidas tras este entrenamiento:

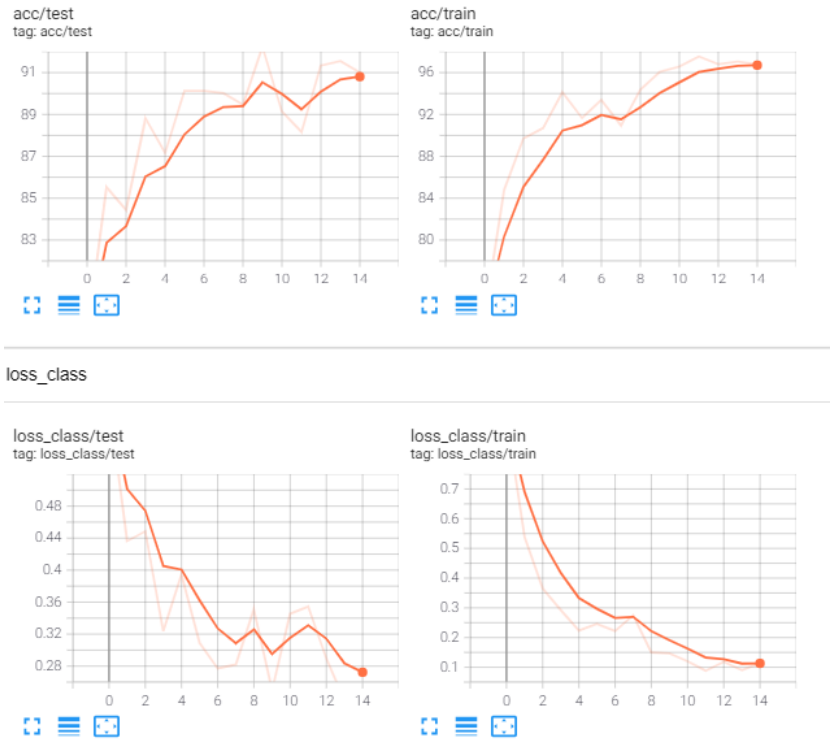


Figura 3.23: Gráficas que muestran los resultados del entrenamiento.

Los gráficos de la parte izquierda de la figura 3.23 corresponden con la etapa de testeo, mientras que los de la derecha con la de entrenamiento. Como se puede ver en ambos casos tras 15 ciclos se alcanzan salidas que superan el 90 % de aciertos, con pérdidas inferiores al 0.3 (recordemos que las pérdidas muestran la diferencia entre el resultado obtenido y el ideal).

Finalmente se comprobará el comportamiento de la red al introducir nubes de puntos a mano y se visualizará el resultado en lugar de usar una función de seguimiento. Para ello lo primero que se debe hacer es estudiar más a fondo las salidas de la red. En este caso la red genera un tensor de 10 valores (uno para cada objeto) que muestran la probabilidad de que la nube de puntos represente alguno de ellos. Cada posición del tensor hace referencia a un objeto, por lo que la predicción será el índice cuya probabilidad sea mayor. Si se busca dentro de los ficheros de ModelNet se puede encontrar un listado que muestra el número correspondiente a la casilla que hace referencia cada objeto. Este número es el que aparece en el parámetro y que se explicó anteriormente en los datos de entrada. En este caso los valores hacen referencia a los siguientes objetos:

Objeto	Índice
Bañera	0
Cama	1
Silla	2
Escritorio	3
Armario	4
Monitor	5
Mesilla de noche	6
Sofá	7
Mesa	8
Retrete	9

Tabla 3.1: Salidas de la red entrenada con ModelNet 10.

Para poder leer las salidas se desarrolla el siguiente código, encargado de descifrar cual es objeto que presenta una mayor probabilidad:

```
output=output.cpu()                                  #Pasamos la salida a la cpu
array_output=output.detach().numpy().tolist() #Convertimos la salida a lista
max_value=max(array_output)                          #Buscamos el valor más alto
predicción=array_output.index(max_value)       #El resultado será el índice
```

De esta forma se crea un traductor capaz entender las salidas de la red. Ahora para verificar si ha acertado se debe comparar el resultado obtenido con el parámetro *y (groundtruth)* de la nube de puntos. Para ello se añade al traductor lo siguiente:

```
objeto_real=muestra.cpu()
objeto_real=objeto_real.y.detach().numpy().tolist()
if (objeto_real[0] == predicción):
    print('Resultado acertado')
else:
    print('Resultado fallado')
```

Juntando ambos códigos crea una función *descifrador* que será la utilizada para comprobar el funcionamiento de la red. Finalmente se añade el siguiente código para visualizar los resultados:

```
#Selección del punto {run: "auto"}
Objeto_1 = 1
Objeto_2 = 3
Objeto_3 = 6
Objeto_4 = 9

objetos_id=[Objeto_1,Objeto_2,Objeto_3,Objeto_4]

p = pv.Plotter(notebook=True,shape=(1, len(objetos_id)),window_size=[1024,412])

for i, objeto in enumerate(objetos_id):

  #Seleccionamos la ventana
  p.subplot(0, i)

  #Titulamos con el resultado de la red
  subtitulo= descifrador(objeto,output)
  corrección=str(subtitulo[1])
  p.add_title(subtitulo[0], color=corrección)

  #Mostramos el objeto
  sample_1= test_samples[batch_id]
  sample_1 = sample_1.pos[objeto].numpy()
  print('Resultado: ',dataset.test_dataset[0][objetos_id[i]].y[0])
  #print(sample_1)
  point_cloud = pv.PolyData(sample_1)
  point_cloud['y'] = sample_1[:,1]
  p.add_points(point_cloud)

  #Posicionamos la camara
  p.camera_position = [-1,5, -10]

p.show()
```

Figura 3.24: Código del visualizador.

A continuación se muestra su funcionamiento con varios ejemplos, donde se puede ver el comportamiento de la red:

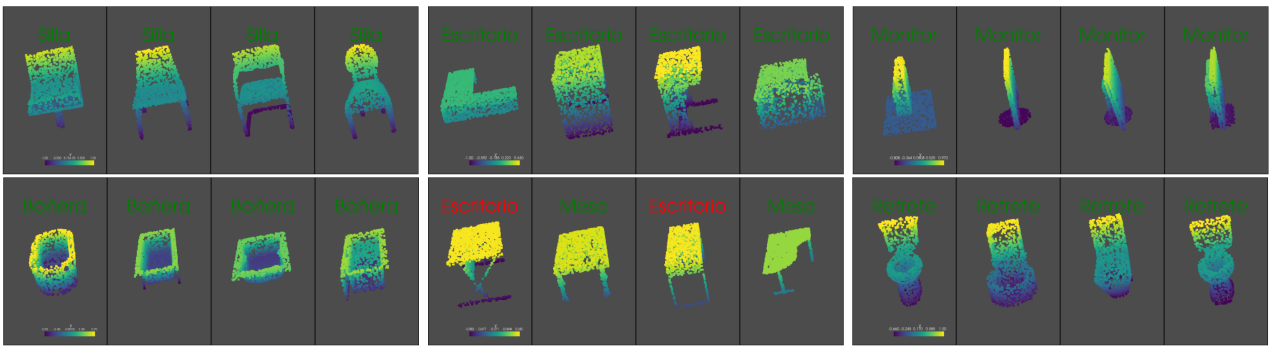


Figura 3.25: Ejemplos de salida.

3.2.5.2 Resultados del notebook de segmentación

Una vez preparados los datos y la estructura de la red se puede pasar a la fase de entrenamiento utilizando la siguiente sentencia:

```
trainer = Trainer(model, dataset)
trainer.fit()
```

Los resultados obtenidos con este notebook no son los esperados. Independientemente de cuanto se entrene la red, siempre se obtienen unos valores bastante elevados de aciertos (entorno al 90 %) según las gráficas, pero al visualizar los resultados la mayoría de veces no corresponde con el *groundtruth*. A modo de ejemplo tras 5 ciclos de entrenamiento las gráficas obtenidas por el *tracker* rondan el 85 % de aciertos, pero al visualizar los resultados se obtiene lo siguiente:

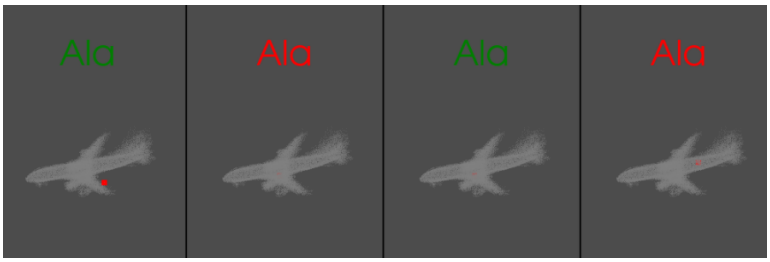


Figura 3.26: Resultados del entrenamiento sobre datos de aviones.

Como se puede apreciar da igual qué punto se estudie que el resultado siempre indica la misma parte (en este caso siempre dice que el punto pertenece al ala) y si se analiza el vector de salida se comprueba que efectivamente la segunda celda es la que siempre tiene un valor más alto con mucha diferencia, mientras que la tercer y cuarta tienen un valor similar pero varias veces más bajo (cuanto más entrenemos más bajo es este valor) y la primera pese a tener un valor más moderado es también varias veces más baja que la segunda. Lo destacable de este punto es cómo se ha analizado este resultado. Se ha una función descifrador muy similar a la del notebook anterior, pero puesto que en este caso la salida de la red es un tensor de tamaño variable se han tenido que tener en cuenta varias consideraciones:

```
def descifrador(clases,output,real_output):
  #print('Clases: ',clases)
  #print('Output: ',output)

  #Primero traducimos la clase diccionario la cual nos da las posiciones en las que se vana  guardar las predicciones
  diccionario=clases
  indices=list(diccionario.values())
  nombre=list(diccionario.keys())
  #print('Objeto:', nombre[0])
  #print('Estos son los indices para tu objeto: ',indices,' por favor, cambia la sentencia de abajo con los valores correspondientes')

  #Avion=[0,1,2,3], Coche=[8,9,10,11]
  if (nombre[0]=='Airplane'):
    index=[0,1,2,3]
  if (nombre[0]=='Car'):
    index=[8,9,10,11]

  #Convertimos el resultado para poder trabajar con el:
  resultado_real=real_output.cpu().detach().numpy().tolist()
  #print('Resultado real: ',resultado_real)

  #Ahora guardaremos los valores en una lista
  valores_predict=[]
  for valor in index:
    salida=output[valor].item()
    valores_predict.append(round(salida,5))
  print('Lista de valores: ',valores_predict)
  #Y obtenemos el índice del valor más alto
  max_value=max(valores_predict)
  predict=valores_predict.index(max_value)
```

Figura 3.27: Primera parte del código para descifrar la salida.

El código para descifrar el resultado en este caso es mucho más complejo ya que dependiendo del

objeto se debe mirar en unas u otras celdas dentro del tensor. Por último una vez obtenido el valor de la predicción se compara con la etiqueta real y se comprueba si el resultado ha sido acertado.

Capítulo 4

Experimentación con el framework

Dicen que la gente puede cambiar, pero... ¿Será eso cierto? Si deciden que quieren volar, ¿les salen alas? No lo creo. No tienes que cambiar tú, sino como haces las cosas. Tienes que crear una manera de volar mientras sigues siendo tú mismo.

Sora Nai

A lo largo de este capítulo se llevarán a cabo diversos experimentos con el fin de explotar las capacidades que nos brinda el framework. En concreto se experimentará partiendo del notebook de clasificación disponible en el Github [10], el cual se estudió a lo largo del anterior capítulo 3. Sobre este se plantearán una serie de modificaciones que se llevarán a cabo aplicando los conocimientos adquiridos en el capítulo anterior. Antes de empezar se ha de destacar que para esta parte el entorno de trabajo cambia. Debido a las restricciones computacionales de Google Colab estos experimentos se llevarán a cabo en remoto gracias a un ordenador de la universidad que cuenta con una unidad de procesamiento de gráficos (GPU).

4.1 Estudio de los hiperparámetros

Lo primero que se llevará a cabo es un estudio de los hiperparámetros presentes en el notebook de clasificación. Se observará cómo afectan dichos parámetros al resultado final de la red, y se estudiará la mejor manera de configurarlos.

4.1.1 Número de ciclos de entrenamiento

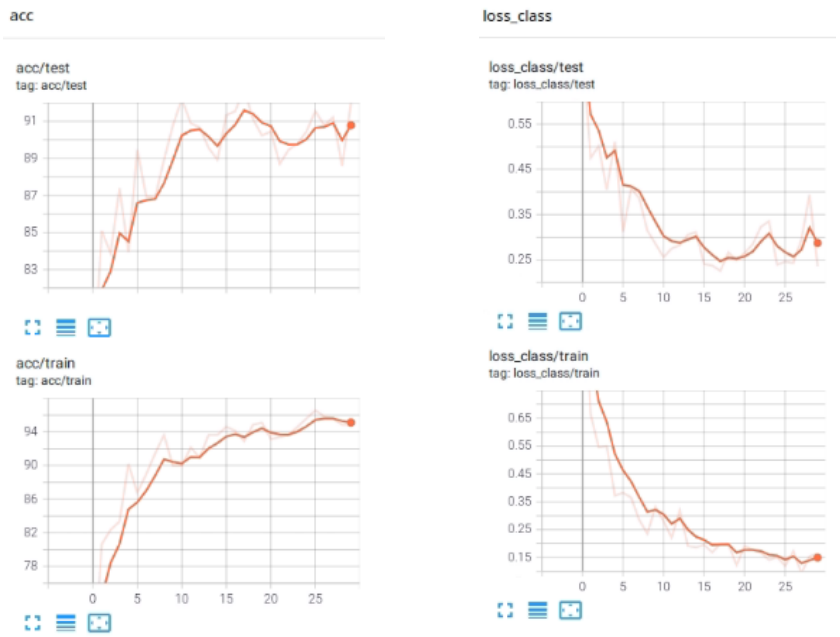
A la hora de entrenar una red neuronal es importante definir un número de ciclos óptimo, ya que si este es demasiado alto se podría llegar a provocar sobre-aprendizaje (*overfitting*) al ajustar demasiado el valor de los pesos a las entradas, y de esta forma perjudicando el comportamiento de la red ya que esta no sería capaz de reconocer datos nuevos. Además, realizar más ciclos el entrenamiento conlleva un aumento tanto en tiempo como en memoria requerida. Por otro lado un número de ciclos demasiado bajo no permitiría alcanzar un modelo capaz de diferenciar entre las diferentes clases.

Para estudiar el efecto que tiene este parámetro se entrenará la red usada en el notebook de clasificación del capítulo 3.2.4.1 variando el número de ciclos (*epoch*) y se comprobaran los resultados:

Batch size	12	12	12	12
Learning rate	0.01	0.01	0.01	0.01
Epoch	10	15	20	30
Resultado final entrenamiento	89.71 %	92.10 %	93.38 %	**94.85 %**
Error final entrenamiento	0.35	0.21	0.21	**0.16**
Mejor resultado entren. | epoch	91.67 % | 7	92.10 % | 15	96.32 % | 18	**96.57 % | 26**
Mínimo error entren. | epoch	0.28 | 7	0.20 | 12	**0.11 | 18**	0.12 | 26
Resultado final testeo	89.91 %	**91.34 %**	**91.34 %**	90.79 %
Error final testeo	0.31	0.26	0.25	**0.24**
Mejor resultado testeo | epoch	89.91 % | 10	91.45 % | 10	91.45 % | 18	**92.76 % | 18**
Mínimo error testeo | epoch	0.28 | 9	0.25 | 10	0.25 | 15	**0.23 | 18**

Tabla 4.1: Resultados del estudio del número de ciclos de entrenamiento.

En base a los datos mostrados en la tabla 4.1 se puede decir que los mejores resultados se obtienen para 30 ciclos de entrenamiento, pero realmente en ese caso se está produciendo *overfitting*. Esto se puede identificar fácilmente atendiendo a las siguientes gráficas que presentan los resultados obtenidos durante la fase de entrenamiento:



(a) Resultados del entrenamiento.

(b) Errores del entrenamiento.

Figura 4.1: Resultados del entrenamiento durante 30 ciclos. A la izquierda (a) el porcentaje de aciertos y a la derecha (b) la gráfica de las pérdidas.

Como se puede observar en la figura 4.1, conforme se aumentan los ciclos los gráficos referentes a los datos de entrenamiento mejoran, teniendo un mayor porcentaje de acierto y menor error. Por otro lado los gráficos referentes al conjunto de testeo llegados a cierto punto empiezan a empeorar. Si se observan los valores de la tabla 4.1 se puede apreciar como se obtuvieron los mejores resultados

para el conjunto de testeo en el decimoctavo ciclo, y a partir de ahí los resultados empeoran. Esto es una muestra clara de sobre-aprendizaje (*overfitting*) ya que la red está memorizando los datos de entrenamiento, y por lo tanto cada vez le cuesta más identificar nuevos modelos.Por ello en base a los resultados obtenidos se puede concluir que un número aceptable de ciclos debe estar comprendido entre [10, 20] ya que por encima se puede encontrar *overfitting*.

De todas forma es imposible concretar un valor óptimo para este hiperparámetro, ya que cada vez que se entrena la red los mejores resultados se obtienen en iteraciones distintas. Por esta razón la mejor forma de trabajar con ello es utilizando un set con datos diferentes a los usados durante la fase de entrenamiento, de forma que se pueda comprobar después de cada iteración los resultados obtenidos con datos con los que no ha entrenado, y así se puede crear un mecanismo de parada automática cuando estos resultados empiecen a empeorar.

4.1.2 Tamaño del batch

A lo largo del capítulo 3.2.2 se expuso como los datos de entrada a las redes son almacenados en paquetes. Estos paquetes se conocen como *batch* y son creados con el fin de mejorar el entrenamiento de la red, ya que al pasar los datos agrupados la velocidad del entrenamiento aumenta y se requiere de menos memoria. Por otro lado esta técnica tiene algunos inconvenientes, ya que si los grupos son demasiado grandes la red puede tener problemas de generalización, y si son muy pequeños la estimación del gradiente puede ser poco precisa.

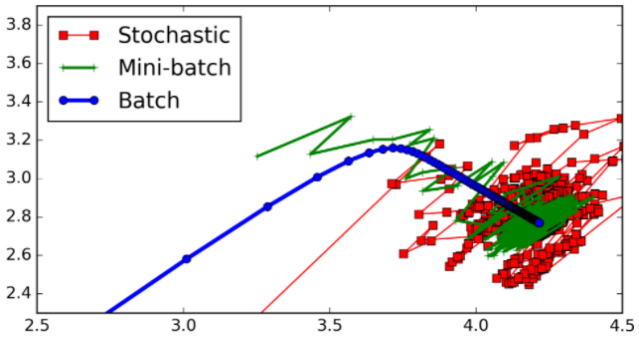


Figura 4.2: Representación gráfica de la dirección del gradiente usando diferentes métodos de agrupamiento.[7]

En el gráfico de la figura 4.2 se puede ver el entrenamiento de una red usando tres métodos de agrupamiento diferentes. La línea roja representa un entrenamiento estocástico sin agrupamiento, la verde representa un entrenamiento con 100 muestras agrupadas de 10 en 10, y por último en la azul se agruparon todas las muestras. Como se puede apreciar, en el entrenamiento usando un único batch el gradiente define un comportamiento más ordenado, ya que este no realiza cambios bruscos, pero para alcanzar su estado óptimo requiere de más ciclos que los otros. Por otro lado en el caso del estocástico puro podemos observar como el comportamiento del gradiente es más caótico, pero en menos ciclos es capaz de alcanzar unos valores cercanos a los finales. El método *mini-batch*

[7]Gráfica extraida de https://stats.stackexchange.com/questions/153531/what-is-batch-size-in-neural-network. Los ejes representan la dirección del primer (eje de abscisas) y segundo (eje de ordenadas) gradiente.

acaba siendo una mezcla entre ambos, requiriendo más ciclos que el estocástico pero concentrando sus valores entorno al estado óptimo.

Para el estudio de este hiperparámetro se debe focalizar en la parte de código que se expuso en el capítulo 3.2.2.1, en la cual se realiza el agrupamiento de los datos. Allí se define el parámetro *batch_size*, el cual determina el número de datos que serán agrupados. A continuación se realizan varias pruebas variando este valor con el fin de comprender el efecto que este tiene sobre el entrenamiento. Las pruebas se realizan manteniendo el *learning rate* a 0.01 y a lo largo de 10 ciclos de entrenamiento (*epoch*). Los resultados obtenidos fueron los siguientes:

Batch size	4	8	12	16
Learning rate	0.01	0.01	0.01	0.01
Epoch	10	10	10	10
Resultado final entrenamiento	74.50 %	88.50 %	89.71 %	**91.25 %**
Error final entrenamiento	0.77	0.40	0.35	**0.29**
Mejor resultado entren. | epoch	74.50 % | 10	90.50 % | 7	91.67 % | 7	**92.25 % | 8**
Mínimo error entren. | epoch	0.77 | 8	0.34 | 7	0.28 | 7	**0.24 | 8**
Resultado final testeo	86.23 %	88.16 %	89.91 %	**91.45 %**
Error final testeo	0.53	0.32	0.31	**0.29**
Mejor resultado testeo | epoch	86.23 % | 10	89.80 % | 9	89.91 % | 10	**91.45 % | 10**
Mínimo error testeo | epoch	0.45 | 8	0.30 | 9	0.28 | 9	**0.25 | 8**

Tabla 4.2: Resultados del estudio del *batch size*.

Como se puede observar en la tabla 4.2 conforme se aumenta el valor de este hiperparámetro se obtienen mejores resultados ya que el porcentaje de aciertos en el entrenamiento y testeo es mayor y el error cometido en ambos es menor. A partir del tamaño 16 la memoria de la GPU se llena, por lo que de este apartado se puede deducir que en estas condiciones el mejor valor con el que trabajar para este hiperparámetro es 16. De todas formas en una GPU de mayor capacidad se podría seguir aumentando dicho valor, por lo que habría que seguir analizando si los resultados siguen mejorando.

4.1.3 Learning rate

El siguiente hiperparámetro que se estudiará es el *learning rate*. Este hiperparámetro define la velocidad con la que se modifica el gradiente al hacer el cálculo de las pérdidas tal y como se vio en el capítulo 2.1.3. Un valor demasiado alto de este hiperparámetro permite al modelo entrenar rápidamente, pero a cambio es posible que no pueda encontrar la solución óptima, ya que lo más probable es que se estanque en una solución intermedia. Por otro lado un valor demasiado bajo pese a poder encontrar mejores soluciones, requiere de mucho más tiempo de entrenamiento. Por esta razón se suelen usar valores comprendidos entre 0.1 y 0.001 tal y como podemos ver en [32]. Este valor se define desde el optimizador, tal y como se vio en el capítulo 3.2.4, y para su estudio se irá variando su valor entre 0.2 y 0.001, manteniendo el tamaño de batch a 16, ya que anteriormente se llegó a la conclusión de que este era su valor óptimo. Para tomar los datos la red se entrena durante 10 ciclos (*epoch*).

Batch size	16	16	16	16
Learning rate	0.2	0.1	0.01	0.001
Epoch	10	10	10	10
Resultado final entrenamiento	**92.00 %**	90.50 %	91.25 %	91.75 %
Error final entrenamiento	**0.21**	0.30	0.29	0.38
Mejor resultado entren. | epoch	**93.25 % | 8**	92.00 % | 8	92.25 % | 8	91.75 % | 10
Mínimo error entren. | epoch	**0.21 | 10**	0.26 | 8	0.24 | 8	0.38 | 9
Resultado final testeo	82.35 %	89.58 %	**91.45 %**	90.10 %
Error final testeo	0.54	0.30	**0.29**	0.34
Mejor resultado testeo | epoch	88.82 % | 8	89.58 % | 10	**91.45 % | 10**	90.24 % | 9
Mínimo error testeo | epoch	0.30 | 8	0.30 | 10	**0.25 | 8**	0.34 | 10

Tabla 4.3: Resultados del estudio del *learning rate*.

En la tabla 4.3 se puede observar que los mejores resultados en el entrenamiento se obtienen para el valor más elevado de *learning rate*, pero esto choca con los resultados obtenidos con las muestras de testeo ya que en este caso también son los peores. Esto indica un principio de *overfitting* ya que la red ha alcanzado su estado final rápidamente, y una vez alcanzado se ha dedicado a memorizar. En cambio con un valor de 0.01 la red ha sido capaz de alcanzar valores similares en el entrenamiento y mejores en el testeo, por lo que no ha se ha visto afectada por el *overfitting*. En base a estas pruebas se puede corroborar lo escrito en [32], y se puede concluir que el valor óptimo para nuestro caso sería de 0.01.

4.1.4 Batch size vs learning rate

Antes de continuar es conveniente ver si existiese una correlación entre el *batch size* y el *learning rate*. Para ello se repetirán los entrenamientos variando ambos parámetros a lo largo de 10 ciclos:

Batch size / Learning rate	**8**	**12**	**16**
0.1	85.50 % | 86.84 %	91.42 % | 85.75 %	90.50 % | 89.58 %
0.01	88.50 % | 88.16 %	89.71 % | 89.91 %	91.25 % | **91.45** %
0.001	87.00 % | 89.14 %	89.71 % | 90.46 %	**91.75** % | 90.10 %

Tabla 4.4: Tabla con los resultados de la comparación *learning rate* y *batch size*. Los datos muestran el porcentaje con el set de entrenamiento | porcentaje set de testeo.

En vista de los resultados expuestos en la tabla 4.4 parece que ambos parámetros no tienen una correlación directa, ya que como era de esperar en base a los visto anteriormente, en todos los casos los porcentajes aumentan conforme lo hace el tamaño del batch y funcionan mejor cuando el valor de *learning rate* es bajo.

4.1.5 Tipo de optimizador

A lo largo de ambos *notebooks* estudiados en el capítulo 3 se habló del optimizador. Los optimizadores son una serie de algoritmos que se encargan de cambiar los atributos de la red neuronal, tales como los pesos y el *learning rate*. En el capítulo 3.2.4 se explicó el uso del optimizador *Adam*, pero además de este hay disponibles otros optimizadores. A continuación se estudiará el comportamiento de algunos optimizadores:

- **Adam** [31]: El algoritmo de optimización *Adam* implementa el promedio móvil exponencial de los gradientes para escalar la tasa de aprendizaje. Es computacionalmente eficiente y requiere de poca memoria lo que lo ha convertido en uno de los algoritmos de optimización más populares.
- **AdaDelta** [33]: El algoritmo de optimización *AdaDelta* es una extensión de *Adagrad* (algoritmo de gradiente adaptativo) en el que se restringe la ventana del gradiente acumulado, pasando a definirla con un tamaño fijo, el cual depende del promedio de pérdidas anterior y del gradiente actual. *Adagrad* es un método de tasa de aprendizaje adaptativo en el cual se realizan actualizaciones más grandes para parámetros poco frecuentes y actualizaciones más pequeñas para parámetros frecuentes.
- **ASGD (Averaged Stochastic Gradient Descent)** [34]: El algoritmo de optimización *ASGD* se trata del algoritmo de optimización estocástico ordinario con el añadido de almacenar la media de sus parámetros en el tiempo. Cuando la optimización termina es este vector de parámetros el que toma el lugar de los pesos.

Optimizador	**Adam**	**AdaDelta**	**ASGD**
Batch size	16	16	16
Learning rate	0.01	0.01	0.01
Epoch	10	10	10
Resultado final entrenamiento	**91.25 %**	80.00 %	84.25 %
Error final entrenamiento	**0.29**	1.08	0.58
Mejor resultado entren. | epoch	**92.25 % | 8**	80.00 % | 10	86.25 % | 9
Mínimo error entren. | epoch	**0.24 | 8**	1.08 | 10	0.54 | 9
Resultado final testeo	**91.45 %**	79.57 %	84.94 %
Error final testeo	**0.29**	1.00	0.51
Mejor resultado testeo | epoch	**91.45 % | 10**	79.57 % | 10	85.64 % | 9
Mínimo error testeo | epoch	**0.25 | 8**	1.00 | 10	0.51 | 10

Tabla 4.5: Resultados del estudio del optimizador.

Como se puede apreciar en los resultados y corroborando la conclusiones del estado del arte, para trabajos con nubes de puntos tridimensionales el optimizador que da mejores resultados es el *Adam*.

4.2 Clasificación de otras bases de datos

Este experimento se basará en la red creada para el notebook de clasificación, estudiada en el capítulo 3.2.3. Se estudiará más a fondo la estructura de los datos de entrada, y se intentará introducir modelos procedentes de otras bases de datos.

4.2.1 Datos de entrada

Para lograr el objetivo final lo primero que se necesita es estudiar qué clase de datos son los que se deben introducir en el modelo. Para ello primero se buscará información dentro de los archivos del framework Pytorch Points 3D [10]. En la carpeta *models* es donde se guardan los scripts referentes a las arquitecturas tal y como se explicó en el capítulo 3.1.1. En este directorio se encuentra el fichero de la arquitectura *RSNet* [27] y en él se encuentra la función de inicialización de la red, donde se especifica que clase de inputs requiere:

```
def set_input(self, data, device):
    """Unpack input data from the dataloader and perform necessary pre-processing steps.
    Parameters:
        input: a dictionary that contains the data itself and its metadata information.
    Sets:
        self.data:
            x -- Features [B, C, N]
            pos -- Features [B, 3, N]
    """
    data = data.to(device)
    if data.x is not None:
        data.x = data.x.transpose(1, 2).contiguous()
    self.input = data
    if data.y is not None:
        self.labels = torch.flatten(data.y).long()  # [B,N]
    else:
        self.labels = data.y
    self.batch_idx = torch.arange(0, data.pos.shape[0]).view(-1, 1).repeat(1, data.pos.shape[1]).view(-1)
    if self._use_category:
        self.category = data.category
```

Figura 4.3: Función de entrada de RSNet [10].

Como se puede leer en el comentario de la figura 4.3 los atributos necesarios para poder pasar una muestra a través de la red son el vector de posición de los puntos **pos** con tres campos *(número de batches, dimensiones (obligatoriamente 3), número de puntos)* y el vector de características **x** con otros 3 campos *(número de batches, características, número de puntos)*. Para comprobarlo se creará un dato de ejemplo con las características necesarias, con el objetivo de garantizar los requisitos mínimos requeridos por la red para obtener un resultado:

```
from torch_geometric.data import Batch, Data

input_ejemplo=[]

batch_size= 2
Points=2048

pos=torch.randn((batch_size,Points,3))
x=torch.ones((batch_size,Points,3))
input_ejemplo.append(Data(pos=pos,x=x))

ex_data= Batch.from_data_list(input_ejemplo)
print(ex_data)

output_ex=model.forward_V2(ex_data)
print(output_ex)
```

```
Batch(batch=[2], pos=[2, 2048, 3], ptr=[2], x=[2, 2048, 3])
tensor([[-2.8823, -2.8787, -2.8822, -2.8772, -2.0579, -1.8816, -2.0143, -1.8776,
         -1.8799, -2.8747],
        [-1.7238, -2.0827, -1.7292, -2.2283, -2.7218, -2.7236, -2.7223, -2.7236,
         -2.7236, -2.4828]], grad_fn=<LogSoftmaxBackward>)
```

Figura 4.4: Código para generar un input de ejemplo para la red RSNet.

Se puede observar en la figura 4.4 que basta con tener un objeto con los vectores de características y posición para que nuestra red sea capaz de predecir, tal y como ponía en el comentario de la figura 4.3 pero al hacer este experimento se ha podido extraer la siguiente información adicional:

- Se requiere de al menos 2 muestras ($batch\_size \geq 2$). Si intentamos poner a 1 el tamaño del batch salta un error, el cual dice que este valor tiene que ser mayor que 1. Cualquier otro número mayor que 1 es aceptado.
- Por la misma razón expuesta arriba, para poder pasar los objetos estos necesariamente han de ir agrupados en batches. No se puede pasar un objeto suelto ya que las dimensiones del tensor darían error.
- Tanto el parámetro y como el parámetro *grid_size* vistos en la figura 3.10 son irrelevantes a la hora de hacer predicciones (siempre y cuando no se realice el cálculo de pérdidas, ya que para este sí que es necesario el atributo y).

En base a esta información si se busca pasar una sola muestra, tomada por ejemplo desde un Lidar, se podría crear un modelo de apoyo tal y como se muestra en la figura 4.4 y juntarlo con la muestra a analizar. Los resultados se obtienen de manera independiente, por lo que solo tendríamos que leer el del objeto real. Esta limitación viene impuesta por el autor bajo la idea de que un solo punto no puede definir un objeto, pero en este caso esta limitación resulta perjudicial. Es posible que modificando alguna línea en el script de RSNet presente en el framework se pueda llegar a trabajar con una sola muestra, pero hay muchos ficheros que hacen referencia a otros, por lo que llegar a realizar estas modificaciones no es tarea sencilla y esta solución aunque sea menos práctica es más simple e igual de efectiva.

Por último se adjunta una imagen de la salida de la red, donde se puede apreciar las muestras de ejemplo que hemos creado y como el modelo ha realizado una predicción. No aporta información relevante, pero junto con lo expuesto en este apartado se puede entender la estructura de las entradas:

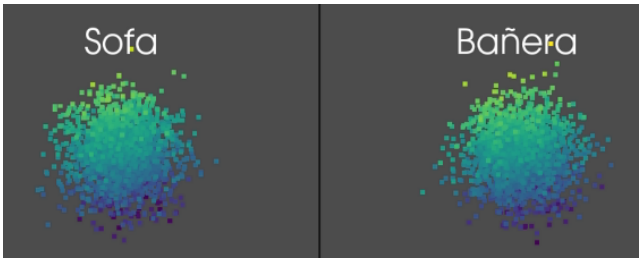


Figura 4.5: Predicción del modelo ante la entrada de ejemplo creada en la figura 4.4.

4.2.2 Base de datos Shapenet

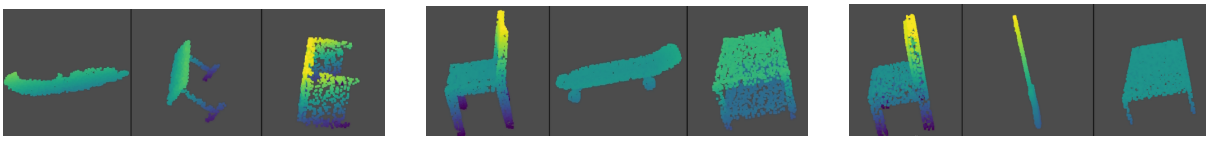
Una vez se ha analizado la estructura de los datos de entrada, lo siguiente que se hará es intentar introducir a la red modelos de otra base de datos. En este caso se usará la base de datos *ShapeNet* [14] ya que esta base de datos ha sido estudiada anteriormente en el capítulo 3.2.1.2. Para cargar la base de datos se usará un *yaml* parecido a los creados en el capítulo 3.2.2. En este caso el archivo

que usaremos como base será el que se creó para el notebook de clasificación en el capítulo 3.2.2.1, ya que es en este donde se definen transformaciones fundamentales para trabajar con *RSNet* como la de *FixedPoints*:

```
#Configure the dataset {run: "auto"}
CATEGORY = "All" #@param ["Airplane", "Car", "Table", "Bag", "All", "Motorbike"] {allow-input: true}
USE_NORMALS = True #@param {type:"boolean"}|

shapenet_yaml = """
class: shapenet.ShapeNetDataset              #Base de datos utilizada
task: classification                           #Tarea a realizar
dataroot: %s
normal: %r                                   # Use normal vectors as features
first_subsampling: 0.02                      # Grid size of the input data
pre_transforms:                              # Offline transforms, done only once
    - transform: NormalizeScale              # Realiza una transformación donde centra y normaliza las posiciones
    - transform: GridSampling3D              # Agrupa los puntos en vóxeles de tamaño "size"
      params:
        size: ${first_subsampling}

    - transform: FixedPoints
      lparams: [2048]

train_transforms:                            # Data augmentation pipeline
    - transform: RandomNoise                 #Aplica ruido a los datos de entrada del entrenamiento
      params:
        sigma: 0.01
        clip: 0.05
    - transform: RandomRotate
      params:
        degrees: 180
        axis: 2

""" % (os.path.join(DIR,"data"), USE_NORMALS)

from omegaconf import OmegaConf
params = OmegaConf.create(shapenet_yaml)
if CATEGORY != "All":
    params.category = CATEGORY
else:
    params.category = None
```

Figura 4.6: Estructura del yaml utilizado para cargar la base de datos Shapenet.

Una vez se han cargados los datos se pueden verificar las transformaciones realizadas en los modelos usando el visualizador:



(a) Muestras de entrenamiento. (b) Muestras de testeo. (c) Muestras de validación.

Figura 4.7: Estructura de la capa de clasificación.

En este caso, ya que la tarea a realizar es de clasificación, se han cargado todas las categorías disponibles en la base de datos. Al hacer esto la estructura generada en la variable es un poco diferente a la que vimos con anterioridad en el capítulo 3.2.2.2. La variable *dataset_shapenet.X_dataset* (donde la *X* representa el conjunto, ya sea entrenamiento, validación o testeo) tendrá dos partes, una con la categoría (avión, coche, gorra, etcétera) y otra con el indicador de la muestra. Al entrar a analizar alguna muestra se puede ver que la estructura es más parecida a la que se estudió con anterioridad en el capítulo 3.2.2.2:

```
print(dataset_shapenet.val_dataset[0])
Data(category=[2048], grid_size=[1], id_scan=[1], origin_id=[2048], pos=[2048, 3], x=[2048, 3], y=[2048])
```

Figura 4.8: Estructura de los datos cargados de Shapenet.

Para ser lo más fieles posibles con lo estudiado hasta ahora, se seguirán los pasos dados durante el notebook de clasificación del capítulo 3. Los datos se cargarán en batches de tamaño 12 usando el módulo *create_dataloader*, el cual fue explicado en el capítulo 3.2.2.1. Luego se separan los batches con el siguiente bucle para tener los datos lo más parecidos a los de la figura 3.10:

```
shapenet_test_loader=dataset_shapenet.test_dataloaders[0]
shapenet_test_samples=[]
device='cuda'

for data in shapenet_test_loader:
  data.to(device)
  shapenet_test_samples.append(data)

print('Máximo batch_id: ',len(shapenet_test_samples))
print('Test: ', shapenet_test_samples[0])
```

```
Máximo batch_id:  240
Test:  SimpleBatch(category=[12, 2048], grid_size=[12, 1], id_scan=[12, 1], origin_id=[12, 2048], pos=[12, 2048,
3], x=[12, 2048, 3], y=[12, 2048])
```

Figura 4.9: Código para extraer los batches.

Se usarán los datos de testeo para realizar las pruebas, pero realmente cualquier conjunto nos hubiese servido puesto que la red no ha entrenado con esta base de datos y por lo tanto no conoce ninguno de ellos. Lo único que se debe tener en cuenta con este conjunto es que la tupla que lo engloba tiene un nivel adicional, por lo que para usarlo se debe poner *shapenet_dataset.test_dataloader[0]*. A continuación se pasará un conjunto de datos por la red para comprobar su comportamiento. Puesto que la red ha sido entrenada con otra base de datos no hay muchos objetos en común, pero en ambas hay modelos de mesas y sillas por lo que se seleccionará un batch en el que aparezca alguno de los dos elementos citados. Puesto que en estos datos el parámetro *y (groundtruth)* tiene una configuración diferente a la esperada para realizar el entrenamiento, se utilizará la función *forward_V2* explicada en el capítulo 3.2.3.1, la cual no realiza el cálculo de pérdidas:

```
output=(model.forward_V2(shapenet_test_samples[batch_id])).cpu()
```

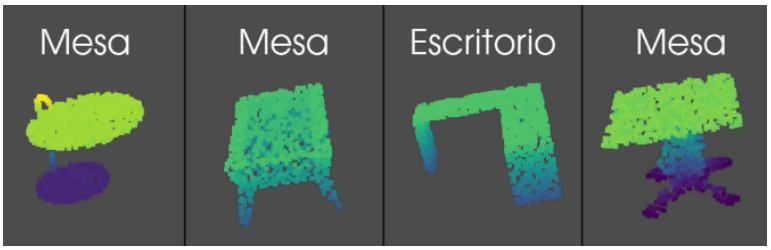


Figura 4.10: Resultados obtenidos al pasar los modelos de Shapenet por la red entrenada en ModelNet.

Como se puede observar en la figura 4.10, la red es capaz de trabajar perfectamente con datos de otras bases de datos. De hecho en la imagen 4.10 se puede observar como un modelo catalogado como *mesa* en Shapenet es identificado como un *escritorio* ya que se asemeja bastante a los escritorios que se encontraban en ModelNet.

4.3 Tarea de clasificación usando modelos de ShapeNet

El siguiente paso amplia el anterior en el que se han pasado diferentes datos a través de la red. En este caso se entrenará la red directamente con la base de datos de Shapenet.

4.3.1 Modificación de la base de datos Shapenet

Como se vio en la sección anterior 4.2 los datos cargados de Shapenet tienen una configuración poco práctica para trabajar con la red *RSNet* ya que esta se ha especializado en tareas de clasificación mientras que los datos se han preparado para realizar tareas de segmentación. De hecho, se han podido pasar las muestras gracias a que se creó la función *forward_V2()*, la cual no realiza el cálculo de las pérdidas, pues para poder hacer eso necesitamos mirar el parámetro y (resultados deseados) de los datos y en el caso de ShapeNet este módulo indica a que parte del objeto hace referencia cada punto, lo cual es distinto de lo esperado. Para entenderlo mejor se muestra a continuación una comparativa entre ambas estructuras:

```
print('Test sample',dataset.test_dataset[0][1])
print('Shapenet sample',dataset_shapenet.test_dataset[0][1])
```

```
Test sample Data(grid_size=[1], pos=[2048, 3], x=[2048, 3], y=[1])
Shapenet sample Data(category=[2048], grid_size=[1], id_scan=[1], origin_id=[2048], pos=[2048, 3], x=[2048, 4], y=
[2048])
```

Figura 4.11: Comparación de la estructura de los datos de entrada creados con la base de datos ModelNet y ShapeNet.

En la imagen se muestra la estructura de dos muestras arbitrarias, una extraída de la base de datos ModelNet (base de datos para la que está preparada la red) y la otra de la base de datos ShapeNet. Como se observa en el caso de ModelNet al estar preparado para trabajar en tareas de clasificación cada muestra contiene los puntos del modelo y una etiqueta que lo clasifica, o sea el parámetro y tiene un único valor. En el caso de ShapeNet al estar preparado para ser usado en tareas de segmentación las muestras contienen los puntos del modelo y las etiquetas son las partes a las que hace referencia cada punto, por eso el parámetro y es una lista del mismo tamaño que la cantidad de puntos que tiene el modelo, tal y como se explicó en el capítulo 3.2.2.2. Esta es la razón que dificulta la tarea de entrenar la red con esta base de datos, ya que no se puede llevar a cabo el cálculo de las pérdidas. Para solventar este problema tenemos que customizar los datos para que su estructura coincida con la que necesita la red de clasificación. Para ello lo primero que se debe hacer es buscar el fichero encargado de cargar la base de datos. Tal y como se vio en el capítulo 3.1.1, este fichero se encuentra en el directorio *datasets*, en concreto dentro de una carpeta llamada *segmentation*. Allí se halla el fichero que se nos proporciona de ejemplo para cargar la base de datos *Shapenet*. Tras su estudio se pueden extraer las siguientes características:

- La clase principal **ShapeNetDataset()** está compuesta por una subclase **ShapeNet()**, la cual se encarga de cargar los datos y se llama una vez por cada conjunto que se quiera cargar (entrenamiento, testeo y validación)
- En la clase principal **ShapeNetDataset()** se definen tres funciones, las cuales son necesarias para poder trabajar con el framework:
  - **class_to_segments()**: Devuelve las categorías en las que se van a segmentar los objetos.
  - **is_hierarchical()**: Devuelve *true* si hay más de una categoría.

  - **get_tracker()**: Carga los archivos necesarios para realizar los seguimientos en la etapa de entrenamiento.

- La subclase **ShapeNet()** es la encargada de descargar y generar la estructura de los datos. La primera vez que esta es llamada descarga los modelos con una estructura determinada, y los guarda en paquetes procesados. Si es llamada nuevamente carga los modelos de estos paquetes.

Una vez se ha entendido la fase de cargado de datos lo que se hará es crear un nuevo fichero de descarga de la base de datos *Shapenet*, pero esta vez se modificará la etapa de generación de la estructura para que los datos se generen de una forma adecuada para trabajar en la tarea de clasificación. Para ello se debe hacer hincapié en la sección que genera la estructura de los datos:

```
def _process_filenames(self, filenames):
    data_raw_list = []
    data_list = []
    categories_ids = [self.category_ids[cat] for cat in self.categories]
    cat_idx = {categories_ids[i]: i for i in range(len(categories_ids))}

    has_pre_transform = self.pre_transform is not None

    id_scan = -1
    for name in tq(filenames):
        cat = name.split(osp.sep)[0]
        if cat not in categories_ids:
            continue
        id_scan += 1
        data = read_txt_array(osp.join(self.raw_dir, name))
        pos = data[:, :3]
        x = data[:, 3:6]

        y = data[:, -1].type(torch.long)
        category = torch.ones(x.shape[0], dtype=torch.long) * cat_idx[cat]
         category = data[:, -1].type(torch.long)
         y = torch.tensor([cat_idx[cat]])

        id_scan_tensor = torch.from_numpy(np.asarray([id_scan])).clone()
        data = Data(pos=pos, x=x, y=y, category=category,
                    id_scan=id_scan_tensor)
        data = SaveOriginalPosId()(data)
        if self.pre_filter is not None and not self.pre_filter(data):
            continue
        data_raw_list.append(data.clone() if has_pre_transform else data)
        if has_pre_transform:
            data = self.pre_transform(data)
            data_list.append(data)
    if not has_pre_transform:
        return [], data_raw_list
    return data_raw_list, data_list
```

Figura 4.12: Función de carga de modelos de la base de datos ShapeNet para tareas de segmentación.

Como se puede apreciar en la figura 4.12, para esta tarea el parámetro y es generado como un tensor que carga los datos sacados de una lista. Como resultado para cada punto tenemos un valor distinto de y, ya que cada punto pertenece a una parte distinta dentro de un objeto. Por otro lado tal y como se vio en el capítulo 3.2.2.2, el parámetro *category* contiene un vector que indica el modelo global que forman todos los puntos. En el nuevo script lo que se hará es modificar el contenido de y, consiguiendo que este parámetro contenga un único valor que indique el objeto que representa el modelo. Para ello basta con modificar la sentencia de la siguiente manera:

```
y=torch.tensor([cat_idx[cat]])
```

Además, se debe cambiar el *tracker* ya que este consta de una serie de sentencias que llevan a cabo el registro de los resultados del entrenamiento. Como ahora se realizará una tarea de clasificación se usará uno especializado para esta función, el cual lo podemos encontrar en la carpeta *metrics*, explicada en el capítulo 3.1.1. Así se crea la siguiente función de seguimiento:

```
from torch_points3d.metrics.classification_tracker
import ClassificationTracker

def get_tracker(self, wandb_log: bool, tensorboard_log: bool):
    return ClassificationTracker(self,
    wandb_log=wandb_log,
    use_tensorboard=tensorboard_log)
```

4.3.2 Entrenamiento de la red RSNet con la base de datos Shapenet

Con estas modificaciones ya se puede entrenar la red *RSNet* para que trabaje con esta base de datos. Para ello se seguirá el esquema definido en el capítulo 3, pero antes de hacer nada se deben borrar los ficheros procesados de *Shapenet* guardados en el equipo, ya que si estos han sido creados con anterioridad, al lanzar el nuevo script no los reemplazará. Lo siguiente será descargar los datos de *Shapenet* mediante el yaml preparado en el capítulo 4.2.2. Para ello se usa la siguiente sentencia:

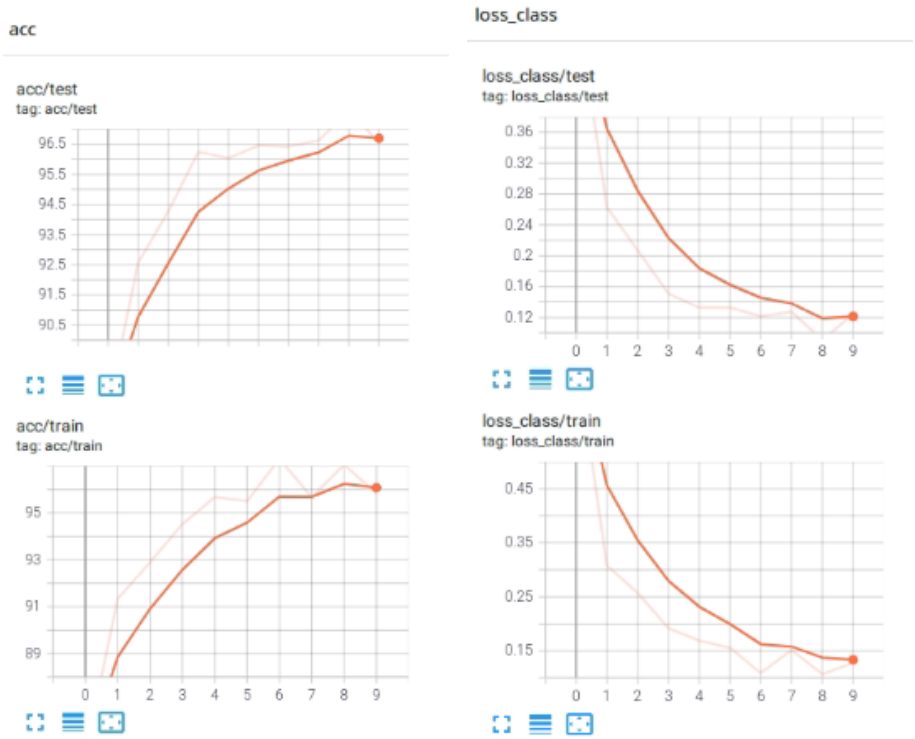
```
from torch_points3d.datasets.classification.shapenet import ShapeNetDataset
dataset_shapenet = ShapeNetDataset(params)
dataset_shapenet
```

Nótese que en este caso la clase *ShapeNetDataset* es importada de la carpeta de clasificación, que es donde se ha guardado el script modificado del capítulo 4.3.1. Como *Shapenet* contiene 16 objetos, en la estructura de la red se debe cambiar el número de salidas. Para ello en el _init_(), a la hora de definir la parte de la estructura *RSNet*, se modifica el número de salidas de la siguiente forma:

```
class RSConvCLassifier(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.encoder = RSConv("encoder", input_nc= 3 * USE_NORMAL,output_nc = int(16), num_layers=4)
        self.log_softmax = torch.nn.LogSoftmax(dim=-1)
```

Figura 4.13: Creación de la red para trabajar con 16 salidas.

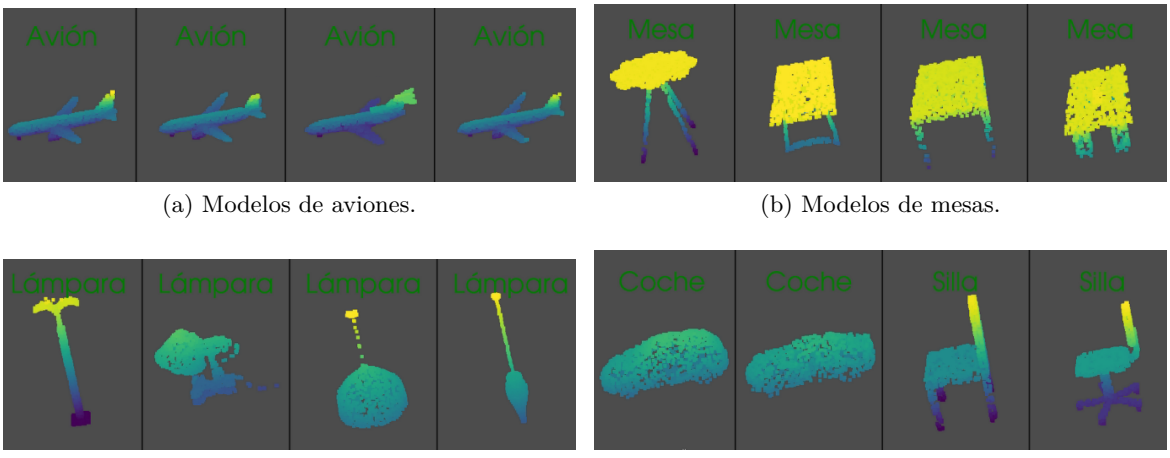
Una vez se han cargado los datos y creado el modelo, los siguientes pasos son realizar la separación en paquetes tal y como explicamos en el capítulo 3.2.2.2 y entrenar la red al igual que en el capítulo 3.2.4.2. Los resultados que se han obtenido del entrenamiento se pueden observar en la figura 4.14.



(a) Resultados del entrenamiento. (b) Pérdidas en el entrenamiento.

Figura 4.14: Resultados de entrenamiento de la red RSNet en la base de datos Shapenet.

Como se puede ver en la figura tras el entrenamiento de la red alcanza unos resultados que rondan el 95 % de aciertos, con pérdidas por debajo de 0.2. Podemos visualizar estos resultados pasando muestras de validación a través de la red, tal y como se muestra en la figura 4.15.



(a) Modelos de aviones. (b) Modelos de mesas.

(c) Modelos de lámparas. (d) Modelos de coches y sillas.

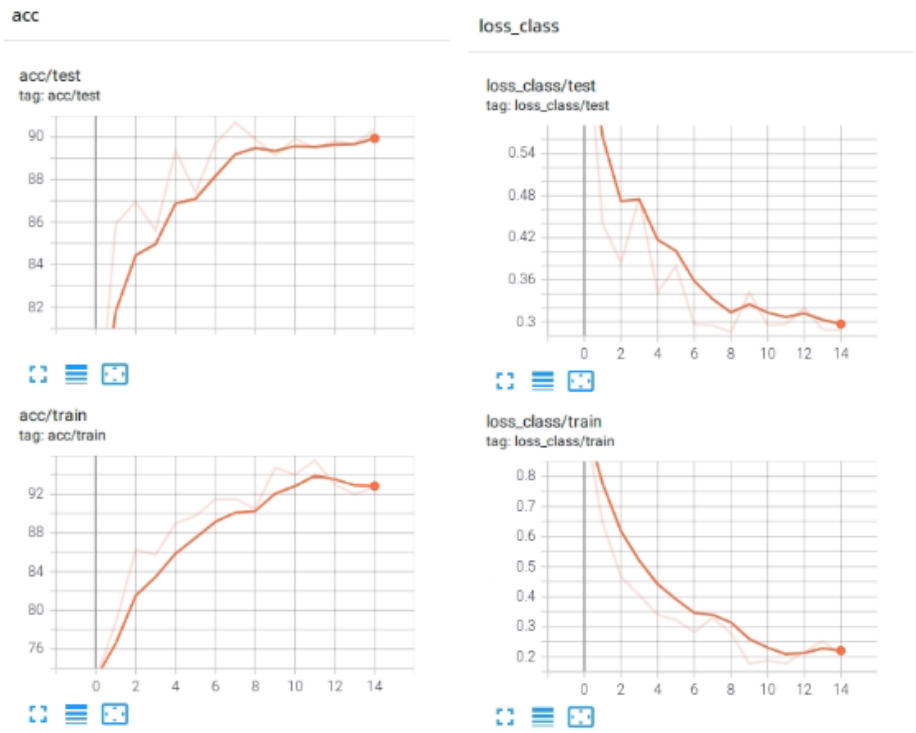
Figura 4.15: Visualización de resultados obtenidos al pasar muestras de validación de la base de datos Shapenet.

4.4 Transfer learning entre ModelNet y Shapenet

Finalmente se llevará a cabo un experimento basado en la técnica de *transfer learning*. Para ello se hará uso de una red RSNet entrenada en la base de datos ModelNet, y se reajustarán los pesos usando la base de datos Shapenet.

4.4.1 Entrenamiento con ModelNet

Para la primera fase se llevará a cabo un entrenamiento del modelo en la base de datos ModelNet tal y como se explicó en los capítulos 3.2.2.1 y 3.2.3.1. Los resultados de este entrenamiento se pueden observar en la figura 4.16.



(a) Resultados del entrenamiento. (b) Pérdidas en el entrenamiento.

Figura 4.16: Resultados de entrenamiento de la red RSNet en la base de datos Modelnet.

Como se puede apreciar en la figura 4.16, la red ha sido entrenada durante 15 ciclos, alcanzando unos porcentajes de acierto cercanos al 90 %, con pérdidas rondando el 0,3. Una vez entrenada la red se guardará su estado haciendo uso de la librería *Pytorch*:

```
torch.save(model.state_dict(), 'checkpoint.pth')
```

Esta sentencia nos permite guardar los valores de los pesos que encontramos en la red en un archivo *.pth* de forma que podemos reiniciar la memoria de la GPU para borrar tanto los modelos cargados de ModelNet como el resto de variables temporales.

4.4.2 Modificaciones en la red

Lo primero que se hace es cargar el modelo guardado anteriormente y congelar el valor de los pesos, ya que no queremos que estos cambien. Para ello usamos las siguientes sentencias:

```
#Cargado del modelo:
model2 = RSConvCLassifier()
model2.load_state_dict(torch.load('checkpoint.pth'))

#Congelamiento de los pesos:
for param in model2.encoder.parameters():
    param.requires_grad=False
```

Para llevar a cabo el **transfer learning** se debe entender la estructura de la red para poder adaptarla a la nueva base de datos. La red está compuesta por los siguientes niveles:

- **encoder**: En esta parte se hereda la estructura de la red RSNet. Está compuesta por cuatro módulos:
  - **down_modules**: Compuesta por cuatro niveles formados por capas *OriginalRSConv()*, las cuales a su vez están formadas por una secuencia de capas de normalización *BatchNorm2d()* y *BatchNorm1d()*.
  - **inner_modules**: Capa *GlobalDenseBaseModule()* formada por una capa *Conv2D()*.
  - **up_modules**: Capa con una lista de módulos.
  - **mlp**: Compuesta por una capa *Seq()* con una única capa *Conv1D()*. Esta a su vez está compuesta por tres subcapas:
    - Conv1d(): capa convolucional de una dimensión con 512 valores de entrada y 10 de salida.
    - BatchNorm1d(): capa de normalización de 10 valores de salida.
    - LeakyReLU(): capa ReLU con una pendiente negativa de 0.01
- **log_softmax**: Esta es la capa de clasificación final que implementamos a la hora de crear la red en el capítulo 3.2.3.1. Se trata de una única capa *Softmax()* de dimensiones *(dim=-1)* (o sea ajustada al número de entradas).

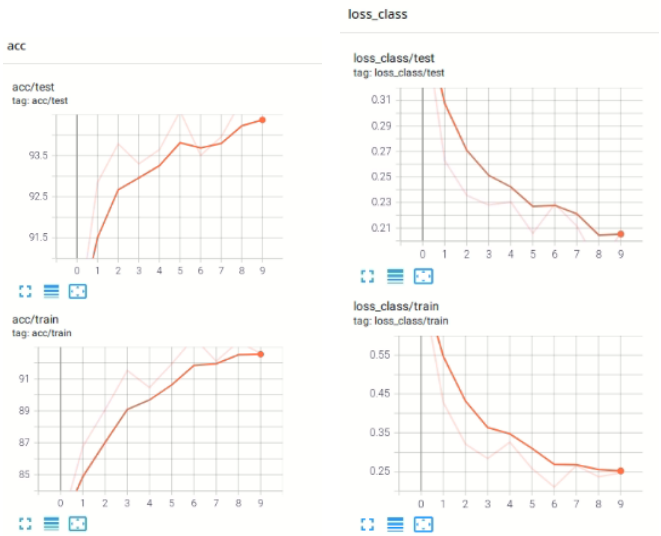
En base a esta información se debe cambiar la capa *mlp* de la parte del *encoder*, ya que es esta la que define la salida de la red. Como se puede observar esta etapa tiene 10 valores de salida ya que la base de datos ModelNet contaba con 10 modelos, pero la nueva base de datos Shapenet cuenta con 16. Para cambiar esta capa lo que se hará es reemplazarla por otra con las dimensiones deseadas. Para ello en la carpeta *core* explicada en el capítulo 3.1.1 se encuentran los scripts con la estructura de esas capas:

```
from torch_points3d.core.common_modules.dense_modules import Conv1D
from torch_points3d.core.common_modules.base_modules import Seq

layer_mlp=Seq()
layer_mlp.append(Conv1D(512,16, bn=True, bias=False))

model2.encoder.mlp=layer_mlp
```

De esta forma se obtiene una red preentrenada en ModelNet pero con una estructura idónea para trabajar con Shapenet. Para finalizar se carga la base de datos Shapenet tal y como se hizo en el capítulo 4.3.1, y se entrena la red en ella.

4.4.3 Resultados del entrenamiento

Finalmente se lleva a cabo el entrenamiento de la red tal y como se hizo en el capítulo 3.2.4.2. En este caso se entrena la red durante 10 ciclos, obteniendo los siguientes resultados:

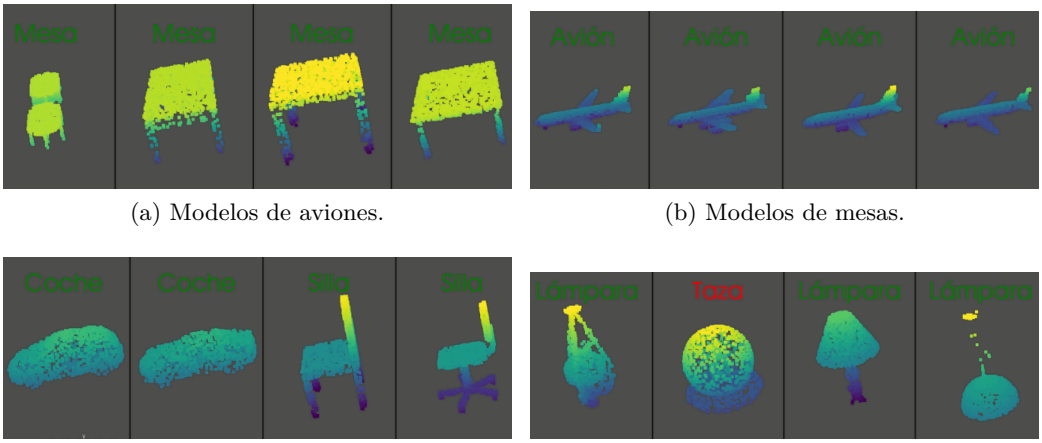


(a) Resultados del entrenamiento.

(b) Pérdidas en el entrenamiento.

Figura 4.17: Resultados de entrenamiento de la red basada en RSNet en la base de datos Shapenet.

Como se puede apreciar en la figura 4.17, al haber sido entrenada con anterioridad, la red rápidamente es capaz de alcanzar valores de aciertos muy elevados, sobrepasando el 90 % en pocos ciclos. Para finalizar se adjuntan algunos resultados de ejemplo:



(a) Modelos de aviones.

(b) Modelos de mesas.

(c) Modelos de lámparas.

(d) Modelos de coches y sillas.

Figura 4.18: Visualización de resultados obtenidos al pasar muestras de validación de la base de datos Shapenet.

En este caso realizar el entrenamiento de esta manera resulta beneficioso. La principal ventaja que se puede observar es en términos de tiempo, ya que entrenar la red con la base de datos ModelNet toma aproximadamente 1:30 minutos por iteración, mientras que con Shapenet este tiempo aumenta a 2:30 minutos. Además si observamos las gráficas de la figura 4.17 podemos ver como en pocas iteraciones ya supera el 90 % de aciertos, lo que puede resultar fundamental en caso de contar con menor capacidad de GPU.

Por otro lado también es cierto que en el apartado en el que entrenamos directamente sobre la base de datos Shapenet (sección 4.3.2) los resultados obtenidos fueron aproximadamente un 5 % mejores. Por ello para decidir si hacer uso de esta técnica se debe realizar un balance entre tiempo y memoria contra resultados.

Capítulo 5

Conclusiones y líneas futuras

Finalmente se llevará a cabo un análisis del trabajo realizado y se sacará una conclusión sobre el estado actual del framework y sobre los resultados obtenidos. Posteriormente se añadirán una serie de ideas que pueden derivar de este trabajo.

5.1 Conclusiones

A lo largo de este trabajo se ha realizado un estudio en profundidad de las posibilidades que ofrece el framework *Pytorch Points 3D*, así como su aplicación en un ejemplo práctico.

El autor proporciona un entorno de trabajo para realizar diversas tareas de extracción de información de nubes de puntos aplicando redes neuronales convolucionales, y de cierta forma el framework cumple su cometido. Pese a ello, al trabajar con el framework y adaptarlo a nuevas necesidades, se hace patente que existen ciertos aspectos poco pulidos dentro del mismo. El principal problema que se puede citar es la falta de información, ya que al llevar poco tiempo activo apenas se puede encontrar documentación, y la poca que hay resulta insuficiente en la mayoría de situaciones. Sin embargo, gracias a los ejemplos se puede extraer mucha información tal y como se ha visto en este trabajo. Otro problema que cabe mencionar es que resulta muy complejo el implementar cualquier experimento que requiera de una modificación de algún parámetro ya establecido. Hacer cualquier cambio requiere de una investigación exhaustiva por la estructura del framework hasta encontrar algún fichero del cual se pueda extraer la suficiente información para poder comprender y modificar dicho parámetro.

Pese a estos problemas el framework puede resultar de utilidad en múltiples tareas gracias a su forma de trabajar. Si se busca implementar alguna de las redes presentes en el mismo, hacerlo con la ayuda del framework puede resultar beneficioso ya que cuentan con múltiples métodos integrados, como por ejemplo el *tracker* para el análisis de resultados. De igual forma descargar las bases de datos se hace sencillo una vez se entienden los ficheros correspondientes, y con un poco de manejo de los mismos se pueden llegar a conseguir las estructuras de los datos necesarias para realizar cualquier tarea. Además visto que el autor pretende ampliar las prestaciones del mismo es de esperar que en el futuro se puedan encontrar muchas más bases de datos y redes disponibles, así como una documentación ampliada que facilite la introducción al framework.

Respecto a los resultados obtenidos en los experimentos, entrenar redes es un trabajo que requiere tiempo y resulta costoso computacionalmente hablando, por lo que aplicar técnicas como el *transfer learning* puede resultar realmente útil. Por ejemplo, la base de datos Shapenet cuenta con un mayor número de modelos, y por lo tanto cada ciclo de entrenamiento dura más tiempo. Usando la técnica de *transfer learning* se ha podido reducir el número de ciclos necesarios para alcanzar resultados cercanos al 90 %, por lo que en esta ocasión resulta ventajoso entrenar de esta forma ya que requiere de una inversión menor de tiempo y memoria.

Si se analizan los resultados de los hiperparámetros en general las conclusiones a las que se ha llegado son las esperadas. Se ha comprobado que los valores pequeños de *learning rate* pese a necesitar más tiempo de entrenamiento obtienen los mejores resultados. También se ha podido analizar como afectan los ciclos de entrenamiento, pudiendo observar el fenómeno de *overfitting*. Aumentar el tamaño del *batch* ha resultado beneficioso para el entrenamiento, pero por las limitaciones del ordenador no se ha podido estudiar el efecto a mayor escala. La conclusión más llamativa ha sido la del optimizador, ya que aunque era esperable que el optimizador *Adam* diese los mejores resultados, estos han sido mucho mejor que los del resto de optimizadores y la diferencia obtenida ha sido mayor de lo esperado. Finalmente, se ha comprobado que la teoría de la relación entre el *batch size* y el *learning rate* era equívoca.

5.2 Líneas Futuras

Actualmente debido a la falta de información, este trabajo puede servir de guía a la hora de realizar cualquier trabajo en el que esté involucrado el framework ya que se explican su uso en profundidad. Sin embargo, sería posible continuar el trabajo de diversas maneras:

- A partir del capítulo 3 *('Estudio del framework´)* se puede seguir desarrollando una documentación en la que se detallen diversos aspectos que no han entrado en este trabajo. Por ejemplo implementando modelos de otras bases de datos u otras arquitecturas no presentes en el framework.
- En este trabajo se han analizado dos ejemplos (clasificación y segmentación). Sin embargo el framework puede ser utilizado para otras tareas. En el futuro se podría seguir estudiando como aplicarlo en tareas de segmentación panóptica, registro o detección de objetos.
- Siguiendo con el estudio de los hiperparámetros se podrían estudiar como afectan estos a los otros ejemplos por si en ellos alguno actuara de manera diferente o analizar otros parámetros de la configuración.
- Finalmente, para el uso del framework en un problema real se podría implementar la red creada para la clasificación de objetos en la vida real capturando nubes de puntos por medio de dispositivos tipo Lidar o escáner 3D.

Bibliografía

[1] W. McCulloch y W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bulletin of mathematical biophysics, vol. 5*, pp. 115–133, 1943, [Online]. Disponible en: https://www.cs.cmu.edu/~./epxing/Class/10715/reading/McCulloch.and.Pitts.pdf visitado por última vez el [2021-08-21].

[2] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," in *Psychological Review, 65(6)*, 1958, pp. 386–408.

[3] M. Minsky y S. Papert, *Perceptrons: an introduction to computational geometry.* MIT Press, 1969.

[4] D. Rumelhart, G. Hinton, y R. Williams, "Learning representations by back-propagating errors," in *Nature, Vol. 323*, 1986, pp. 533–536.

[5] J. Lighthill, "Artificial intelligence: A paper symposium," 1973, [Online]. Disponible en: http://www.aiai.ed.ac.uk/events/lighthill1973/lighthill.pdf visitado por última vez el [2021-08-21].

[6] A. Krizhevsky, I. Sutskever, y G. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems Vol. 5*, pp. 1097–1105, 2012, [Online]. Disponible en: https://papers.nips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf visitado por última vez el [2021-06-07].

[7] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, y A. Rabinovich, "Going deeper with convolutions." Proceedings of the IEEE conference on computer vision and pattern recognition, 2015, [Online]. Disponible en: https://arxiv.org/pdf/1409.4842.pdf visitado por última vez el [2021-06-07].

[8] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, y D. Hassabis, "Mastering chess and shogi by selfplay with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017, [Online]. Disponible en: https://arxiv.org/pdf/1712.01815.pdf visitado por última vez el [2021-06-10].

[9] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, y D. Amodei, "Language

models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020, [Online]. Disponible en: https://arxiv.org/pdf/2005.14165.pdf visitado por última vez el [2021-06-10].

[10] T. Chaton, N. Chaulet, S. Horache, y L. Landrieu, "Torch-points3d: A modular multi-task framework for reproducible deep learning on 3d point clouds," in *2020 International Conference on 3D Vision (3DV)*, 2020, enlaces:

Paper: https://arxiv.org/pdf/2010.04642.pdf visitado por última vez el [2021-06-10].

Github: https://github.com/nicolas-chaulet/torch-points3d visitado por última vez el [2021-06-10].

Documentacion: https://torch-points3d.readthedocs.io/en/latest/ visitado por última vez el [2021-06-10].

[11] Fey, Matthias, Lenssen, y J. E., "Fast graph representation learning with PyTorch Geometric," *arXiv preprint arXiv:1903.02428*, 2019, enlaces:

Paper: https://arxiv.org/pdf/1903.02428.pdf visitado por última vez el [2021-07-07].

Web: https://pytorch-geometric.readthedocs.io/en/latest/ visitado por última vez el [2021-07-07].

Github: https://github.com/rusty1s/pytorch_geometric visitado por última vez el [2021-07-07].

[12] A. Dai, A. X. Chang, M. Savva, M. Halber, T. Funkhouser, y M. Nießner, "Scannet: Richly-annotated 3d reconstructions of indoor scenes." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017, pp. 5828–5839, enlaces:

Paper: https://arxiv.org/pdf/1702.04405.pdf visitado por última vez el [2021-06-17].

Web: http://www.scan-net.org/ visitado por última vez el [2021-06-17].

Github: https://github.com/ScanNet/ScanNet visitado por última vez el [2021-06-17].

[13] I. Armeni, A. Sax, A. R. Zamir, y S. Savarese, "Joint 2d-3d-semantic data for indoor scene understanding," *arXiv preprint arXiv:1702.01105*, 2017, enlaces:

Paper: https://arxiv.org/pdf/1702.01105.pdf visitado por última vez el [2021-06-17].

Web: http://buildingparser.stanford.edu/dataset.html visitado por última vez el [2021-06-17].

[14] A. X. Chang, T. Funkhouser, L. Guibas, P. Hanrahan, Q. Huang, Z. Li, S. Savarese, M. Savva, S. Song, H. Su, J. Xiao, L. Yi, y F. Yu, "Shapenet: An information-rich 3d model repository," *arXiv preprint arXiv:1512.03012*, 2015, enlaces:

Paper: https://arxiv.org/pdf/1512.03012.pdf visitado por última vez el [2021-06-17].

Web: https://shapenet.org/ visitado por última vez el [2021-06-17].

[15] J. Behley, M. Garbade, A. Milioto, J. Quenzel, S. Behnke, C. Stachniss, y J. Gall, "Semantickitti: A dataset for semantic scene understanding of lidar sequences." IEEE/CVF International Conf. on Computer Vision (ICCV), 2019, enlaces:

Paper: https://arxiv.org/pdf/1904.01416.pdf visitado por última vez el [2021-06-17].

Web: http://www.semantic-kitti.org/ visitado por última vez el [2021-06-17].

[16] A. Zeng, S. Song, M. Nießner, M. Fisher, J. Xiao, y T. Funkhouser, "3dmatch: Learning local geometric descriptors from rgb-d reconstructions." IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017, enlaces:

Paper: https://arxiv.org/pdf/1603.08182.pdf visitado por última vez el [2021-06-17].

Web: http://3dmatch.cs.princeton.edu/ visitado por última vez el [2021-06-17].

[17] S. Fontana, D. Cattaneo, A. L. Ballardini, M. Vaghi, y D. G. Sorrenti, "A benchmark for point clouds registration algorithms," *Robotics and Autonomous Systems Vol. 140*, p. 103734, 2021, enlaces:

Paper: https://arxiv.org/pdf/2003.12841.pdf visitado por última vez el [2021-06-17].

Web: https://github.com/iralabdisco/point_clouds_registration_benchmark visitado por última vez el [2021-06-17].

[18] F. Pomerleau, M. Liu, F. Colas, y R. Siegwart, "Challenging data sets for point cloud registration algorithms," in *International Journal of Robotic Research, vol. 31, no. 14*, 2012, p. 1705–1711.

[19] C. Tong, D. Gingras, K. Larose, T. D. Barfoot, y E. Dupuis, "The canadian planetary emulation terrain 3d mapping dataset," in *International Journal of Robotics Research (IJRR)*, 2013, [Online]. Disponible en: https://doi.org/10.1177/0278364913478897 visitado por última vez el [2021-06-17].

[20] J. Sturm, N. Engelhard, F. Endres, W. Burgard, y D. Cremers, "A benchmark for the evaluation of rgb-d slam systems," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012, pp. 573–580, [Online]. Disponible en: https://vision.in.tum.de/data/datasets/rgbd-dataset visitado por última vez el [2021-06-17].

[21] J. Jeong, Y. Cho, Y. Shin, H. Roh, y A. Kim, "Complex urban dataset with multi-level sensors from highly diverse urban environments," *The International Journal of Robotics Research*, 2019, [Online]. Disponible en: https://irap.kaist.ac.kr/dataset/papers/IJRR2019_dataset.pdf visitado por última vez el [2021-06-17].

[22] A. Geiger, P. Lenz, y R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," *Computer Vision and Pattern Recognition (CVPR),(Providence, USA)*, 2012, enlaces:

Paper: http://www.cvlibs.net/publications/Geiger2012CVPR.pdf visitado por última vez el [2021-06-17].

Web: http://www.cvlibs.net/datasets/kitti/eval_odometry.php visitado por última vez el [2021-06-17].

[23] Z. Wu, S. Song, A. Khosla, F. Yu, L. Zhang, X. Tang, y J. Xiao, "3d shapenets: A deep representation for volumetric shapes," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1912–1920, enlaces:

Paper: https://people.csail.mit.edu/khosla/papers/cvpr2015_wu.pdf visitado por última vez el [2021-06-17].

Web: https://modelnet.cs.princeton.edu/ visitado por última vez el [2021-06-17].

[24] H. Thomas, C. R. Qi, J. Deschaud, B. Marcotegui, F. Goulette, y L. J. Guibas, "Kpconv: Flexible and deformable convolution for point clouds," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 6411–6420, [Online]. Disponible en: https://arxiv.org/pdf/1904.08889.pdf visitado por última vez el [2021-06-17].

[25] C. R. Qi, L. Yi, H. Su, y L. J. Guibas, "Pointnet++: Deep hierarchical feature learning on point sets in a metric space," *arXiv preprint arXiv:1706.02413*, 2017, [Online]. Disponible en: https://arxiv.org/pdf/1706.02413.pdf visitado por última vez el [2021-06-17].

[26] C. R. Qi, H. Su, K. Mo, y L. J. Guibas, "Pointnet: Deep learning on point sets for 3d classification and segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 652–660, [Online]. Disponible en: https://arxiv.org/pdf/1612.00593.pdf visitado por última vez el [2021-06-17].

[27] Y. Liu, B. Fan, S. Xiang, y C. Pan, "Relation-shape convolutional neural network for point cloud analysis," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR, Oral Best Paper Finalist)*, 2019, pp. 8895–8904, [Online]. Disponible en: https://arxiv.org/pdf/1904.07601.pdf visitado por última vez el [2021-06-17].

[28] C. Choy, J. Y. Gwak, y S. Savarese, "4d spatio-temporal convnets: Minkowski convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Patter Recognition (CVPR)*, 2019, [Online]. Disponible en: https://arxiv.org/pdf/1904.08755.pdf visitado por última vez el [2021-06-17].

[29] K. He, X. Zhang, S. Ren, y J. Sun, "Deep residual learning for image recognition," in *Computer Vision and Pattern Recognition (CVPR), 2016*, 2016, [Online]. Disponible en: https://arxiv.org/pdf/1512.03385.pdf visitado por última vez el [2021-06-17].

[30] O. Yadan, "Omegaconf documentation," 2019, [Online]. Disponible en: https://omegaconf.readthedocs.io/en/2.1_branch/usage.html visitado por última vez el [2021-07-07].

[31] D. P. Kingma y J. L. Ba, "Adam: a method for stochastic optimization," *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, 2014, [Online]. Disponible en: https://arxiv.org/pdf/1412.6980.pdf visitado por última vez el [2021-07-07].

[32] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," in *Neural networks: Tricks of the trade*, H. Springer, Berlin, Ed., 2012, pp. 437–478, [Online]. Disponible en: https://arxiv.org/pdf/1206.5533.pdf visitado por última vez el [2021-08-08].

[33] M. D. Zeiler, "Adadelta: an adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012, [Online]. Disponible en: https://arxiv.org/pdf/1212.5701.pdf visitado por última vez el [2021-08-09].

[34] B. T. Polyak y A. B. Juditsky, "Acceleration of stochastic approximation by averaging," *SIAM journal on control and optimization Vol. 30 (4)*, pp. 838–855, 1992, [Online]. Disponible en: https://doi.org/10.1137/0330046 visitado por última vez el [2021-08-09].

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá